IRIX Programmer's Reference Manual

Volume II

Version 5.0

Document Number 007-0602-050

IRIX Programmer's Reference Manual
Version 5.0
Document Number 007-0602-050

Silicon Graphics, Inc.
Mountain View, California

TABLE OF CONTENTS

3. Subroutines

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

#include <device.h>

#include <get.h>

#include <gl.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke IRIX system primitives, which are described in Section 2. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3G) These functions constitute the IRIS Graphics Library which are documented in the *Graphics Library User's Guide*. The **–lgl_s** and **–lm** flags should be specified to acess the graphics library. Declarations for these functions may be obtained from the include file *<gl.h>*. *<device.h>* and *<get.h>* define other constants used by the Graphics Library.

(3M) These functions constitute the Math Library, *libm*. The link editor searches this library in response to the **–lm** option to *ld*(1) or *cc*(1). Declarations for these functions may be obtained from the include file *<math.h>*.

(3S) These functions constitute the ''standard I/O package'' (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the include file *<stdio.h>*.

(3B) IRIX supports many 4.3BSD system calls and library routines. In order to get the maximum Berkeley compatibility, use the following compile line:

cc -D_BSD_COMPAT -o prog prog.c -lbsd

-D_BSD_SIGNALS on the compile line specifically selects the Berkeley signal routines, which is a subset of the compatibility specified by **-D_BSD_COMPAT**.

The following 4.3BSD Standard C Library routines in *libbsd* have different arguments or conflicting semantics with the routines in IRIX *libc* having the same names: *chown, dup2, fchown, getgroups, getpgrp, initgroups, setgroups, setpgrp.* To compile and link a program that calls the BSD version of any of these routines, use a command of the form:

cc prog.c –lbsd

See the "BSD Compatibility" section below for more details.

(3N) These functions constitute the 4.3BSD Internet network library. They are in the standard C library.

(3R) RPC services built on top of Sun's Remote Procedure Call protocol. To compile and link a program that calls any of these routines, use a command of the form:

cc prog.c –lrpcsvc –lsun

Note that this library is provided as part of the NFS option package, so it may not be present on all systems.

(3Y) Yellow Pages routines and RPC support routines. This library contains routines that provide a programmatic interface to Sun's Yellow Pages distributed lookup service. The library contains YP versions of standard lookup routines like *getpwent*(3). The routines that implement the RPC protocol also reside in this library. To compile and link a program that calls (3Y) routines, use a command of the form:

cc prog.c –lsun

This library is provided as part of the NFS option package, so it may not be present on all systems.

(3P) These primitives constitute the parallel processing library, *libmpc*. The link editor searches this library in response to the **–lmpc** option to *ld*(1) or *cc*(1). *libmpc* also contains a full version of *libc* with many of the standard functions adapted for parallel processing. The following calls have been single threaded so that multiple shared processes accessing them simultaneously will function correctly: *getc, putc, fgetc, fputc, ungetc, getw, putw, gets, fgets, puts, fputs, fopen, fdopen, freopen, ftell, rewind, setbuf, setvbuf, fclose, fflush, fread, fwrite, fseek, popen, pclose, printf, fprintf, vprintf, vfprintf, scanf, fscanf, opendir, readdir, scandir, seekdir, closedir, telldir, dup2, srand, rand, malloc, free, calloc, realloc, mallopt, acreate, amalloc, afree, acalloc, arealloc, amallopt.* See *usconfig(3P)* for more information on how to alter the behaviour of these routines. No locking/single threading is done until a process first does a *sproc*(2) call.

(3T) These primitives implement a general terminal interface that provides control over asynchronous communications ports.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

BSD COMPATIBILITY

As described in the discussion of Section 3B above,

cc -D_BSD_COMPAT -o prog prog.c -lbsd

selects maximum compatibility with BSD. The **–lbsd** directive specifies that **libbsd.a** be searched before **libc.a**, which selects the BSD versions of functions that reside in both libraries (duplicated because of identical names yet differing semantics or arguments). The routines that fall into this category are listed in the (3B) section above. The BSD versions may also be selected on a case-by-case basis by prefixing the function name with **BSD** when calling it in the program (e.g. *BSDfchown*).

Specifying **-D_BSD_COMPAT** or **-D_BSD_SIGNALS** on the compile line links with the BSD versions of the signal routines (*kill*, *killpg*, *sigblock*, *signal*, *sigpause*, *sigsetmask*, and *sigvec*). The program must include *<signal.h>* or *<sys/signal.h>*. Note that a "#define _BSD_COMPAT" or "#define _BSD_SIGNALS" placed in the source program before the inclusion of the signal header file has the same effect as specifying the corresponding **–D** compile option.

Defining **_BSD_COMPAT** gives the following additional BSD compatibility features over and above that given by **_BSD_SIGNALS**: you get the BSD version of *setjmp*(3) and including *<sys/types.h>* will cause several additional macros and typedefs to be defined (e.g. *major*, *minor*, *makedev* for dealing with device numbers). **_BSD_COMPAT** may affect more things in future releases.

The System V and BSD versions of the directory routines (*opendir*, *seekdir*, etc.) differ greatly; inclusion of *<dirent.h>* at the top of the user program selects the System V versions, *<sys/dir.h>* selects the BSD set. See also *directory*(3C) and *directory*(3B).

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A NULL pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error.

NULL is defined as **0** in <*stdio.h*>; the user can include his own definition if he is not using <*stdio.h*>.

FILES

/usr/lib/libc.a
/usr/lib/libm.a
/usr/lib/libgl.a
/usr/lib/libbsd.a
/usr/lib/libsun.a
/usr/lib/librpcsvc.a
/usr/lib/libmpc.a

SEE ALSO

Graphics Library User's Guide
ar(1), cc(1), ld(1), nm(1), intro(2), stdio(3S), directory(3C), directory(3B).

DIAGNOSTICS

Functions in the Math Library (3M) may return the values 0 (on underflow), $\pm\infty$ (overflow), and **NaN** (on illegal operation) .

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a ''digit'' in a radix-64 notation.

The characters used to represent ''digits'' are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

a64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

a64l scans the character string from left to right, decoding each character as a 6 bit Radix 64 number.

l64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

CAVEAT

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – terminate current process with a core dump

SYNOPSIS

#include <stdlib.h>

int abort (void);

DESCRIPTION

abort does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results. The core dump can then be examined with the source level debugger *dbx*(1).

abort returns the value of the *kill*(2) system call.

SEE ALSO

dbx(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If **SIGABRT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message ''abort – core dumped'' is written by the shell.

NAME

abs – return integer absolute value

SYNOPSIS

#include <stdlib.h>

int abs (int i);

DESCRIPTION

abs returns the absolute value of its integer operand.

SEE ALSO

floor(3M).

CAVEAT

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

NAME

acreate, adelete, amalloc, afree, arealloc, acalloc, amallopt, amallinfo – arbitrary arena main memory allocator

SYNOPSIS

#include <malloc.h>

void *acreate (void *addr, size_t len, int flags,
struct usptr_s *ushdr, void *(*grow)(size_t, void *));

void *amalloc (size_t size, void *ap);

int adelete (void *ap);

void afree (void *ptr, void *ap);

void *arealloc (void *ptr, size_t size, void *ap);

void *acalloc (size_t nelem, size_t elsize, void *ap);

int amallopt (int cmd, int value, void *ap);

struct mallinfo amallinfo (void *ap);

DESCRIPTION

The arena malloc package provides a main memory allocator based on the *malloc(3X)* memory allocator. This allocator has been extended in two ways: 1) an arbitrary memory space ("arena") may be set up as an area from which to malloc, and 2) the entire package has been semaphored to allow multiple processes to be single threaded while allocating out of the same arena.

Calls to the arena malloc package differ from calls to the standard *malloc(3X)* only in that an arena pointer must be supplied. This arena pointer is returned by a call to *acreate.*

acreate sets up an area defined as starting at virtual address *addr* and extending for *len* bytes. Arenas may be either growing or non-growing and either shared or unshared. An arena that is non-growing is constrained to use only up to *len* bytes of memory. The *grow* parameter should be **NULL** in this case. If the arena is growable, *len* specifies the original size (this MUST be a minimum of 1K bytes) and the *grow* parameter specifies a function that will be called when the allocator requires more memory. The function will be called with two parameters, the number of bytes required and a pointer to the arena requiring the space. The number of bytes requested will always be a multiple of **M_BLKSZ** (see *amallopt* below).

Since the allocator package involves a two-tiered allocation strategy (small blocks and large blocks), various anomalies (such as not being able to allocate any space!) can arise when using very small non-growable arenas (*len* less than 16K). For this reason *acreate* will set **M_BLKSZ** to 512 and

M_MXFAST to 0 for all arenas whose size is less than 16k and is non-growable. These default values may be overwritten via *amallopt.* Users creating very small growable arenas may likewise have to tune the resulting arena's parameters.

If the arena is to be shared amoung multiple processes, then the **MEM_SHARED** flag should be passed, and *ushdr* must be a pointer to a semaphore allocation header as returned from *usinit*(3P). Calling *acreate* with the **MEM_SHARED** flag simply causes *acreate* to allocate a lock, which it then uses to single thread all accesses to the arena. It is the callers responsibility to ensure that the arena is accessible by all processes, and to provide a mechanism to exchange the addresses returned by *amalloc* amoungst the various processes.

adelete causes any resources allocated for the arena (e.g. semaphores) to be freed. Nothing is done with the arena memory itself. No further calls to any arena functions should be made after calling *adelete*.

amalloc and *afree* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "libmpc.a", and is loaded if the option "–lmpc" is used with *cc*(1) or *ld*(1).

amalloc returns a pointer to a block of at least *size* bytes. Requests for greater than *maxfast* bytes are always quad-word aligned; smaller sizes are aligned according to *grain* (see *amallopt* below).

The argument to *afree* is a pointer to a block previously allocated by *amalloc*; after *afree* is performed this space is made available for further allocation, and its contents are destroyed (see *amallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *amalloc* is overrun or if some random number is handed to *afree*.

arealloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

acalloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

amallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST Set *maxfast* to *value*. The algorithm allocates all blocks at or below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 28.

M_NLBLKS Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN Set *grain* to *value*. Requests less than or equal to *maxfast* will be rounded up to the nearest multiple of 4 bytes. The allocated space will be aligned on a "*grain* - (size modulo *grain*)" boundary and will be constrained to be contained within the minimal number of *grain*-sized blocks. For example, if *grain* is 16, requests for 17-20 bytes are rounded to 20 bytes, aligned to 4 bytes, and contained within two 16-byte-aligned blocks; requests for 21-24 bytes are rounded to 24 bytes, aligned to 8 bytes, and contained within two 16-byte-aligned blocks. *Value* will be rounded up to a multiple of the default when *grain* is set. *Grain* must be greater than 0. The default value of *grain* is 16.

M_KEEP Preserve data in a freed block until the next *amalloc*, *realloc*, or *acalloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

M_DEBUG Turns debug checking on if *value* is not equal to 0, otherwise turns debug checking off. When debugging is on, each call to *amalloc* and *afree* causes the entire malloc arena to be scanned and checked for consistency. This option may be invoked at any time. Note that when debug checking is on, the performance of *malloc* is reduced considerably.

M_BLKSZ When *amalloc* requires additional space, it call the defined *grow* funtion. It always requests enough for the current *amalloc* request rounded up to a minimum block size. By default that block size is 8K and may be set to any value greater than 512. If a lot of space is to be allocated, setting the block size larger can cut down on the system overhead. This option may be invoked at any time.

M_MXCHK By default, *malloc* trades off time versus space - if anywhere in the arena there is a block of the appropriate size, *malloc* will find and return it. If the arena has become fragmented due to many *mallocs* and *frees* , it is possible that *malloc* will have to search through many blocks to find one of the appropriate size. If the arena is severely fragmented, the average *malloc* time can be on the order of tens of milliseconds (as opposed to a normal average of tens of microseconds). This option allows the user to place a limit

on the number of blocks that *malloc* will search through before allocating a new block of space from the system. Small values (less than 50) can cause much more memory to be allocated. Values around 100 (the default) cause very uniform response time, with a small space penalty. This option may be invoked at any time.

These values are defined in the <*malloc.h*> header file.

amallopt may be called repeatedly, but, for most commands, may not be called after the first small block is allocated.

amallinfo provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo {
    int arena;          /* total space in arena */
    int ordblks;        /* number of ordinary blocks */
    int smblks;         /* number of small blocks */
    int hblkhd;         /* space in holding block headers */
    int hblks;          /* number of holding blocks */
    int usmblks;        /* space in small blocks in use */
    int fsmblks;        /* space in free small blocks */
    int uordblks;       /* space in ordinary blocks in use */
    int fordblks;       /* space in free ordinary blocks */
    int keepcost;       /* space penalty if keep option */
                        /* is used */
}
```

This structure is defined in the <*malloc.h*> header file. The structure is zero until after the first space has been allocated from the arena.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X), usinit(3P), usnewlock(3P), usmalloc(3P).

DIAGNOSTICS

acreate will return NULL and set *errno* if either *len* is less than 1K or the **MEM_SHARED** flag is passed but *ushdr* is **NULL**. *amalloc*, *arealloc* and *acalloc* return a NULL pointer if there is not enough available memory. On the first call to *amalloc*, *arealloc*, or *acalloc* -1 may be returned and *errno* set if the MEM_SHARED flag is set and it is impossible to allocate a lock. When *arealloc* returns NULL, the block pointed to by *ptr* is left intact. If *amallopt* is called after any allocation (for most *cmd* arguments) or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

NAME

asinh, acosh, atanh – inverse hyperbolic functions

SYNOPSIS

#include <math.h>

double asinh(double x);

double acosh(double x);

double atanh(double x);

DESCRIPTION

These functions compute the designated inverse hyperbolic functions for real arguments.

ERROR (due to Roundoff etc.)

These functions inherit much of their error from *log1p* described in *exp*(3M).

DIAGNOSTICS

Acosh returns the default quiet NaN if the argument is less than 1.

Atanh returns the default quiet NaN if the argument has absolute value bigger than or equal to 1.

SEE ALSO

math(3M), exp(3M)

NAME

assert – program verification

SYNOPSIS

#include <assert.h>

assert(expression);

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be true at this point in the program. If the expression is false (0), it prints a diagnostic comment to standard output and exits via *abort(3C),* leaving a core dump. Compiling with the *cc*(1) option **–DNDEBUG** effectively deletes *assert* from the program.

DIAGNOSTICS

'Assertion failed: file *f* line *n*.' *F* is the source file and *n* the source line number of the *assert* statement.

NAME

barrier, new_barrier, init_barrier, free_barrier – barrier functions

C SYNOPSIS

#include <ulocks.h>

barrier_t *new_barrier (usptr_t *handle);

void free_barrier (barrier_t *b);

void init_barrier (barrier_t *b);

void barrier (barrier_t *b, unsigned n);

FORTRAN SYNOPSIS

integer*4 function new_barrier (handle)
integer*4 handle

subroutine free_barrier (b)
integer*4 b

subroutine barrier (b, n)
integer*4 b
integer*4 n

DESCRIPTION

These routines provide a simple rendezvous mechanism for shared address processes.

new_barrier takes a usptr_t as an argument to indicate the shared arena from which to allocate the barrier. The usptr_t is a previously allocated handle obtained through a call to *usinit(3P)*.

The *barrier* function takes a pointer to a previously allocated and initialized barrier structure (as returned by *new_barrier*) and the number of processes/sub-tasks to wait for. As each process enters the barrier, it spins (busy wait) until all *n* processes enter the barrier. At that time all are released and continue executing.

free_barrier releases all storage associated with *b*.

init_barrier resets the barrier to its default state.

new_barrier will fail if one or more of the following are true:

[ENOMEM] There is not enough space to allocate a barrier structure.

[ENOMEM] It is not possible to allocate a lock.

SEE ALSO

sproc(2), usinit(3P), ussetlock(3P), usunsetlock(3P), usnewlock(3P).

DIAGNOSTICS

Upon successful completion, *new_barrier* returns a pointer to a barrier struct. Otherwise, a value of 0 is returned to the calling process.

NAME

j0, j1, jn, y0, y1, yn – bessel functions

SYNOPSIS

```
#include <math.h>

double j0(double x);

double j1(double x);

double jn(int n, double x);

double y0(double x);

double y1(double x);

double yn(int n, double x);
```

DESCRIPTION

j0 and *j1* return Bessel functions of *x* of the first kind of orders zero and one, respectively. *jn* returns the Bessel function of *x* of the first kind of order *n*.

y0 and *y1* return Bessel functions of *x* of the second kind of orders zero and one, respectively. *yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return a quiet NaN.

BUGS

Arguments too large in magnitude cause *j0*, *j1*, *y0*,and *y1* to return zero with no indication of the total loss of precision.

SEE ALSO

math(3M)

NAME

bsearch – binary search a sorted table

SYNOPSIS

#include <stdlib.h>

```
void *bsearch (const void *key, const void *)base,
        size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nmemb* is the number of elements in the table. *Size* is the size of the key in bytes (*sizeof (*key)*). *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>

#define  TABSIZE        1000

struct node {                   /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];     /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
```

```
                .
                .
                .
                node.string = str_space;
                while (scanf("%s", node.string) != EOF) {
                        node_ptr = (struct node *)bsearch((char *)(&node),
                                   (char *)table, TABSIZE,
                                   sizeof(struct node), node_compare);
                        if (node_ptr != NULL) {
                                (void)printf("string = %20s, length = %d\n",
                                        node_ptr->string, node_ptr->length);
                        } else {
                                (void)printf("not found: %s\n", node.string);
                        }
                }
        }
        /*
                This routine compares two nodes based on an
                alphabetical ordering of the string field.
        */
        int
        node_compare(node1, node2)
        char *node1, *node2;
        {
                return (strcmp(
                                ((struct node *)node1)->string,
                                ((struct node *)node2)->string));
        }
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although *bsearch* is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

bcopy, bcmp, blkclr, bzero – byte string operations

SYNOPSIS

bcopy(src, dst, length)
void *src, *dst;
int length;

bcmp(b1, b2, length)
void *b1, *b2;
int length;

bzero(b, length)
void *b;
int length;

blkclr(b, length)
void *b;
int length;

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

Bcopy copies *length* bytes from string *src* to the string *dst*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero and *blkclr* place *length* zero bytes in the string *b*.

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

#include <sys/types.h>
#include <netinet/in.h>

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines such as the IRIS-4D series, these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostbyname*(3N) and *getservent*(3N).

SEE ALSO

gethostbyname(3N), getservent(3N)

NAME

cfgetospeed, cfgetispeed, cfsetospeed, cfsetispeed – POSIX baud rate primitives

#include <termios.h>

speed_t cfgetospeed (struct termios *termios_p);

int cfsetospeed (struct termios *termios_p, speed_t speed);

speed_t cfgetispeed (struct termios termios_p);

int cfsetispeed (struct termios *termios_p, speed_t speed);

DESCRIPTION

These interfaces are provided for getting and setting the values of the input and output baud rates in the *termios* structure (defined in *<termios.h>*). The effects on the terminal device described below do not become effective until the *tcsetattr* function is successfully called.

Baud Rate Values, declared in *<sys/termio.h>*:

B0	Hang Up	**B600**	600 baud
B50	50 baud	**B1200**	1200 baud
B75	75 baud	**B1800**	1800 baud
B110	110 baud	**B2400**	2400 baud
B134	134 baud	**B4800**	4800 baud
B150	150 baud	**B9600**	9600 baud
B200	200 baud	**B19200**	19200 baud
B300	300 baud	**B38400**	38400 baud

cfgetospeed returns the output baud rate stored in the *termios* structure pointed to by *termios_p*.

cfsetospeed sets the output baud rate stored in the *termios* structure pointed to by *termios_p* to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

cfgetispeed returns the input baud rate stored in the *termios* structure.

cfsetispeed sets the input baud rate stored in the *termios* structure to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate.

DIAGNOSTICS

Upon successful completion all return 0.

cfgetispeed and *cfgetospeed* cannot fail.

When *cfsetispeed* and *cfsetospeed* fail they return –1 and set *errno* as

follows:

[EINVAL] an invalid *speed* value was specified.

SEE ALSO

tcsetattr(3T).

NAME

clock – report CPU time used

SYNOPSIS

#include <time.h>

clock_t clock (void);

DESCRIPTION

clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2), *pclose*(3S), or *system*(3S).

The resolution of the clock is 10 milliseconds on IRIS workstations.

SEE ALSO

times(2), wait(2), popen(3S), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

conv: toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

#include <ctype.h>

int toupper (int c);

int tolower (int c);

int _toupper (int c);

int _tolower (int c);

int toascii (int c);

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from −1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower*, are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

copysign, drem, finite, logb, scalb – copysign, remainder, exponent manipulations

SYNOPSIS

#include <math.h>

double copysign(double x, double y);

double drem(double x, double y);

int finite(double x);

double logb(double x);

double scalb(double x, int n);

DESCRIPTION

These functions are required for, or recommended by the IEEE standard 754 for floating–point arithmetic.

copysign(x,y) returns x with its sign changed to y's.

drem(x,y) returns the remainder r := x – n*y where n is the integer nearest the exact value of x/y; moreover if $|n-x/y| = 1/2$ then n is even. Consequently the remainder is computed exactly and $|r| \le |y|/2$. But *drem(x,0)* is exceptional; see below under DIAGNOSTICS.

finite(x) = 1 just when $-\infty < x < +\infty$,
= 0 otherwise (when $|x| = \infty$ or x is *NaN*)

logb(x) returns x's exponent n, a signed integer converted to double–precision floating–point and so chosen that $1 \le |x|/2^{**}n < 2$ unless x = 0 or $|x| = \infty$ or x lies between 0 and the Underflow Threshold; see below under "BUGS".

scalb(x,n) = x*(2**n) computed, for integer n, without first computing 2**n.

DIAGNOSTICS

IEEE 754 defines *drem(x,0)* and *drem(∞,y)* to be invalid operations that produce a *NaN*.

IEEE 754 defines *logb*(±∞) = +∞ and *logb(0)* = –∞, and requires the latter to signal Division–by–Zero.

SEE ALSO

floor(3M), math(3M).

AUTHOR

Kwok–Choi Ng

BUGS

IEEE 754 currently specifies that

logb(denormalized no.) = *logb*(tiniest normalized no. > 0)

but the consensus has changed to the new proposed standard 854. Almost every program that assumes 754's specification will work correctly if *logb* follows 854's specification instead.

IEEE 754 requires *copysign(x,Nan)* = ±x but says nothing about the sign of a NaN.

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, ignored)
char *block;
int ignored;

DESCRIPTION

crypt is the password encryption function. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

Key is a user's typed password. *Salt* is a two-character string chosen from the set **[a-zA-Z0-9./]**; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Ignored* is unused by *encrypt* but it must be present.

SEE ALSO

getpass(3C), passwd(4).
login(1), passwd(1) in the *User's Reference Manual.*

CAVEAT

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid (char *s);

DESCRIPTION

ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, cftime, ascftime, strftime, tzset – convert date and time to string

SYNOPSIS

```
#include <time.h>
char *ctime (const time_t *clock);
struct tm *localtime (const time_t *clock);
struct tm *gmtime (const time_t *clock);
char *asctime (const struct tm *tm);
int cftime(char *buf, char *fmt, time_t *clock);
int ascftime (char *buf, char *fmt, struct tm *tm);
size_t strftime (char *buf, size_t maxsize, const char *fmt,
                 const struct tm *tm);
extern long timezone, altzone;
extern int daylight;
extern char *tzname[2];
void tzset (void);
```

DESCRIPTION

ctime, *localtime*, and *gmtime* accept arguments of type **time_t,** pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970. *ctime* returns a pointer to a 26-character string in the following form. All the fields have constant width.

Fri Sep 13 00:00:00 1986\n\0

localtime and *gmtime* return pointers to "tm" structures, described below. *localtime* corrects for the main time zone and possible alternate ("Daylight Savings") time zone; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the UNIX system uses.

asctime converts a "tm" structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the "tm" structure, are in the <*time.h*> header file. The structure declaration is:

```
struct  tm {
        int     tm_sec; /* seconds after the minute — [0, 59]  */
        int     tm_min; /* minutes after the hour — [0, 59] */
        int     tm_hour;        /* hour since midnight — [0, 23] */
        int     tm_mday;        /* day of the month — [1, 31] */
        int     tm_mon; /* months since January — [0, 11] */
        int     tm_year;        /* years since 1900 */
        int     tm_wday;        /* days since Sunday — [0, 6] */
        int     tm_yday;        /* days since January 1 — [0, 365] */
        int     tm_isdst;       /* flag for daylight savings time */
};
```

tm_isdst is non-zero if the alternate time zone is in effect.

cftime and *ascftime* provide the capabilities of *ctime* and *asctime*, respectively, as well as additional ones. *cftime* takes an integer of type *time_t* pointed to by *clock* and converts it to a character string. *ascftime* takes a pointer to a "tm" structure and converts it to a character string. In both functions, the characters are placed into the array pointed to by *buf* (plus a terminating \0) and the value returned is the number of such characters (not counting the terminating \0). *fmt* controls the format of the resulting string.

fmt is a character string that consists of field descriptors and text characters, reminiscent of *printf*(3S). Each field descriptor consists of a % character followed by another character which specifies the replacement for the field descriptor. All other characters are copied from *fmt* into the result. The following field descriptors are supported:

%%	same as %
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%d	day of month (01 - 31)
%D	date as %m/%d/%y
%e	day of month (1-31; single digits are preceded by a blank)
%h	abbreviated month name
%H	hour (00 - 23)
%I	hour (00 - 12)
%j	day number of year (001 - 366)
%m	month number (01 - 12)
%M	minute (00 - 59)
%n	same as \n
%p	ante meridian or post meridian
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00 - 59)
%t	insert a tab
%T	time as %H:%M:%S
%U	week number of year (01 - 52), Sunday is the first day of week
%w	weekday number (Sunday = 0)
%W	week number of year (01 - 52), Monday is the first day of week

%x	Local specific date format
%X	Local specific time format
%y	year within century (00 - 99)
%Y	year as ccyy (e.g. 1986)
%Z	time zone name

The difference between %U and %W lies in which day is counted as the first of the week. Week number 01 is the first week with four or more January days in it.

The example below shows what the values in the "tm" structure would look like for Thursday, August 28, 1986 at 12:44:36 in New Jersey.

ascftime (buf, "%A %m %d %j", tm)

This example would result in the buffer containing "Thursday Aug 28 240".

If *fmt* is (char *)0, the value of the environment variable **CFTIME** is used. If **CFTIME** is undefined or empty, a default format is used. The default format string is taken from the file that contains the date and time strings associated with the then current language [see below for details on changing the current language and *cftime*(4) for a description of the structure of these files].

The user can request that the output of *cftime* and *ascftime* be in a specific language by setting the environment variable **LANGUAGE** to the desired language. If **LANGUAGE** is empty, unset or set to an unsupported language, the last language requested will be used (the default is the **usa-english** strings).

strftime is just like *ascftime*, but with the additional *maxsize* parameter, which specifies the size of the character array pointed to by *buf*.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and the main time zone; the external **long** variable *altzone* contains the difference, in seconds, between GMT and the alternate time zone; both, *timezone* and *altzone* default to 0 (GMT). The external variable *daylight* is non-zero if an alternate time zone exists. The time zone names are contained in the external variable *tzname*, which by default is set to

char *tzname[2] = { "GMT", "   " };

The functions know about the peculiarities of this conversion for various time periods for the U.S.A (specifically, the years 1974, 1975, and 1987). The functions will handle the new daylight savings time starting with the first Sunday in April, 1987.

tzset uses the contents of the environment variable **TZ** to override the value of the different external variables. The syntax of **TZ** can be described as follows:

TZ	→	*zone*
	/	*zone signed_time*
	/	*zone signed_time zone*
	/	*zone signed_time zone dst*
zone	→	*letter letter letter*
signed_time	→	*sign time*
	/	*time*
time	→	*hour*
	/	*hour : minute*
	/	*hour : minute : second*
dst	→	*signed_time*
	/	*signed_time ; dst_date , dst_date*
	/	*; dst_date , dst_date*
dst_date	→	*julian*
	/	*julian / time*
letter	→	*a / A / b / B / ... / z / Z*
hour	→	*00 / 01 / ... / 23*
minute	→	*00 / 01 / ... / 59*
second	→	*00 / 01 / ... / 59*
julian	→	*001 / 002 / ... / 366*
sign	→	*− / +*

tzset scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the setting for New Jersey in 1986 could be

"EST5EDT4;117/2:00:00,299/2:00:00" .

or simply

EST5EDT

A southern hemisphere setting such as the Cook Islands could be

"KDT9:30KST10:00;64/5:00,303/20:00"

When the longer format is used, the variable must be surrounded by double quotes as shown. For more details, see *timezone*(4) and *environ*(5). In the longer version of the New Jersey example of **TZ**, *tzname[0]* is EST, *timezone* will be set to 5*60*60, *tzname[1]* is EDT, *altzone* will be set to 4*60*60, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM, and *daylight* will be set to non-zero. Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and

the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be midnight. The effects of *tzset* are thus to change the values of the external variables *timezone*, *altzone*, *daylight* and *tzname*. *tzset* is called by *localtime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set to the correct value by default when the user logs on, via the local /etc/profile file [see *profile*(4)].

FILES

/lib/cftime – directory that contains the language specific printable files

SEE ALSO

time(2), stime(2), gettimeofday(3B), getenv(3C), putenv(3C), printf(3S), cftime(4), profile(4), timezone(4), environ(5).

CAVEAT

The return values for *ctime*, *localtime* and *gmtime* point to static data whose content is overwritten by each call.

Setting the time during the interval of change from *timezone* to *altzone* or vice versa can produce unpredictable results.

The system administrator must change the Julian start and end days annually if the full form of the **TZ** variable is specified.

NAME

isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii, tolower, toupper, toascii, _tolower, _toupper, setchrclass – character handling

SYNOPSIS

#include <ctype.h>

int isdigit (int c);

int isxdigit (int c);

int islower (int c);

int isupper (int c);

int isalpha (int c);

int isalnum (int c);

int isspace (int c);

int iscntrl (int c);

int ispunct (int c);

int isprint (int c);

int isgraph (int c);

int isascii (int c);

int tolower (int c);

int toupper (int c);

int toascii (int c);

int _tolower (int c);

int _toupper (int c);

int setchrclass (char *chrclass);

DESCRIPTION

The character classification macros listed below return nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined on valid members of the character set and on the single value **EOF** [see *stdio*(3S)] (guaranteed not to be a character set member).

isdigit tests for the digits 0 through 9.

isxdigit tests for any character for which *isdigit* is true or for the letters *a* through *f* or *A* through *F*.

islower tests for any lowercase letter as defined by the character set.

isupper tests for any uppercase letter as defined by the character set.

isalpha tests for any character for which *islower* or *isupper* is true and possibly any others as defined by the character set.

isalnum tests for any character for which *isalpha* or *isdigit* is true.

isspace tests for a space, horizontal-tab, carriage return, newline, vertical-tab, or form-feed.

iscntrl tests for ''control characters'' as defined by the character set.

ispunct tests for any character other than the ones for which *isalnum*, *iscntrl*, or *isspace* is true or space.

isprint tests for a space or any character for which *isalnum* or *ispunct* is true or other ''printing character'' as defined by the character set.

isgraph tests for any character for which *isprint* is true, except for space.

isascii tests for an ASCII character (a non-negative number less than 0200.)

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower if the character is one for which *isupper* is true and there a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper if the character is one for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

toascii turns off the bits that are not part of the ASCII character set.

_tolower returns the lowercase representation of a character for which *isupper* is true, otherwise undefined.

_toupper returns the uppercase representation of a character for which *islower* is true, otherwise undefined.

The conversion macros have the same functionality of the functions on valid input, but the macros are faster because they do not do range checking.

All the character classification macros and the conversion functions and macros do a table lookup.

setchrclass initializes the table used by these functions and macros to a specific character classification set. *setchrclass* uses the value of its argument or the value of the environment variable **CHRCLASS** as the name of the datafile containing the information for the desired character set. These datafiles are searched for in the special directory /lib/chrclass.

If *chrclass* is (char *)0, the value of the environment variable **CHRCLASS** is used. If **CHRCLASS** is not set or is undefined, the table retains its current value, which at initialization time is **ascii**.

FILES

/lib/chrclass – directory containing the datafiles for *setchrclass*

SEE ALSO

chrtbl(1M), stdio(3S), ascii(5), environ(5).

DIAGNOSTICS

If the argument to any of the character handling macros is not in the domain of the function, the result is undefined.

If *setchrclass* does not successfully fill the table, the table will not change (initially ''ascii'') and −1 is returned. If everything works, *setchrclass* returns 0.

NAME

curses – terminal screen handling and optimization package

SYNOPSIS

The *curses* manual page is organized as follows:

In SYNOPSIS
- compiling information
- summary of parameters used by *curses* routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:
- An overview of how *curses* routines should be used

In ROUTINES, descriptions of each *curses* routines, are grouped under the appropriate topics:
- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Color Manipulation
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Then come sections on:
- ATTRIBUTES
- COLORS
- FUNCTION KEYS
- LINE GRAPHICS

cc [flag ...] file ... **–lcurses** [library ...]

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.)

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes
int (*init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (*putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y
short pair, color, f, r, g, b

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate()
beep()
box(win, vertch, horch)
can_change_color()
cbreak()
clear()

clearok(win, bf)
clrtobot()
clrtoeol()
color_content(color, &r, &g, &b)
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol,
dmaxrow, dmaxcol, overlay)"
curs_set(visibility)
def_prog_mode()
def_shell_mode()
del_curterm(oterm)
delay_output(ms)
delch()
deleteln()
delwin(win)
doupdate()
draino(ms)
echo()
echochar(ch)
endwin()
erase()
erasechar()
filter()
flash()
flushinp()
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch()
getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_colors()
has_ic()
has_il()
idlok(win, bf)
inch()
init_color(color, r, g, b)
init_pair(pair, f, b)
initscr()
insch(ch)
insertln()
intrflush(win, bf)

isendwin()
keyname(c)
keypad(win, bf)
killchar()
leaveok(win, bf)
longname()
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl()
nocbreak()
nodelay(win, bf)
noecho()
nonl()
noraw()
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pair_content(pair, &f, &b)
pechochar(pad, ch)

pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
printw(fmt [, arg...])
putp(str)
raw()
refresh()
reset_prog_mode()
reset_shell_mode()
resetty()
restartterm(term, fildes, errret)
ripoffline(line, init)
savetty()
scanw(fmt [, arg...])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_attroff(attrs)
slk_attron(attrs)
slk_attrset(attrs)
slk_clear()
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh()
slk_refresh()
slk_restore()
slk_set(labnum, label, fmt)
slk_touch()
standend()
standout()
start_color()
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)

tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, ..., p9)
tputs(str, count, putc)
traceoff()
traceon()
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

The *curses* routines give the user a terminal-independent method of updating screens with reasonable optimization.

In order to initialize the routines properly, **# include <curses.h>** must be included at the beginning of files that use any *curses* routines. In addition, the routine **initscr**( ) or **newterm**( ) must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**( ) must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), after calling **initscr**( ) you should call "**cbreak**(); **noecho**();" Most programs would additionally call "**nonl**(); **intrflush (stdscr, FALSE); keypad(stdscr, TRUE);**".

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**( ). Windows are referred to by variables declared as **WINDOW ***; the type **WINDOW** is defined in **<curses.h>** to be a structure. These data structures are manipulated with routines described below, among which the most basic are **move**( ) and **addch**( ). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then **refresh**( ) is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad**( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE**,

ACS_HLINE, and **KEY_LEFT.**

Routines that manipulate color on color alphanumeric terminals are new in this release of *curses*. To use these routines **start_color**() must be called, usually right after **initscr**(). Colors are always used in pairs (referred to as color-pairs). A color-pair consists of a foregound color (for characters) and a background color (for the field the characters are displayed on). A programmer initializes a color-pair with the routine **init_pair**(). After it has been initialized, **COLOR_PAIR**(n), a macro defined in **<curses.h>**, can be used in the same ways other video attributes can be used. If a terminal is capable of redefining colors the programmer can use the routine **init_color**() to change the definition of a color. The routines **has_color**() and **can_change_color**() return **TRUE** or **FALSE**, depending on whether the terminal has color capabilities and whether the user can change the colors. The routine **color_content**() allows a user to identify the amounts of red, green, and blue components in an initialized color. The routine **pair_content**() allows a user to find out how a given color-pair is currently defined.

curses also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**() is **curscr**, the next call to **wrefresh**() with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**() is **curscr**, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a ''repaint-screen'' function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425**, then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* will first check *$HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be filled in by **initscr()** with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The integer variables **COLORS** and **COLOR_PAIRS** are also defined in **<curses.h>** and contain, respectively, the maximum number of colors and color-pairs the terminal can support. They are initialized by **start_color()**. The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with **p** require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv()** routines imply a call to **move()** before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always **(0,0)**, not **(1,1)**. The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and **pad** are always of type **WINDOW ***.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE**. (*bf* is always of type **bool**.) The types **WINDOW**, **bool**, and **chtype** are defined in **<curses.h>**. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return **NULL** on error.

Sometimes the description of a routine refers to a second routine. If the routine referred to is prefixed with a **w**, then you should assume that other versions of the second routine behave similarly. For example, the description of **initscr()** refers to **wrefresh()**. This implies that the same result will occur if **refresh()** is called.

Overall Screen Manipulation

WINDOW *initscr()

The first routine called should almost always be **initscr()**. (The exceptions are **slk_init()**, **filter()**, and **ripoffline()**.) This will determine the terminal type and initialize all *curses* data structures. **initscr()** also arranges that the first call to **wrefresh()** will clear the screen. If errors occur, **initscr()** will write an appropriate error message to standard error and exit; otherwise, a pointer to

stdscr is returned. If the program wants an indication of error conditions, **newterm**( ) should be used instead of **initscr**(). **initscr**() should only be called once per application.

endwin( )
A program should always call **endwin**() before exiting or escaping from *curses* mode temporarily, to do a shell escape or *system*(3S) call, for example. This routine will restore *tty*(7) modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh**( ) or **doupdate**( ).

isendwin( )
Returns **TRUE** if **endwin**( ) has been called without any subsequent calls to **wrefresh**( ).

SCREEN ***newterm**(type, outfd, infd)
A program that outputs to more than one terminal must use **newterm**( ) for each terminal instead of **initscr**( ). A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm**( ) should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin**( ) for each terminal being used. If **newterm**( ) is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin**( ) is called.

SCREEN ***set_term**(new)
This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh( )
wrefresh (win)

These routines (or **prefresh**( ), **pnoutrefresh**( ), **wnoutrefresh**( ), or **doupdate**( )) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh**( ) copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh**( ) does the same thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)
doupdate( )

These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**–*begin_y* and **cols**–*begin_x*. A new full-screen window is created by calling **newwin(0,0,0,0)**.

mvwin(win, y, x)
Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause any portion of the window to be moved off the screen, it is an error and the window is not moved.

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call **touchwin**() or **touchline**() on *orig* before calling **wrefresh**() on *orig*.

delwin(win)
Delete the named window, freeing up all memory associated with it. If you try to delete a main window before all of its subwindows have been deleted, ERR will be returned.

WINDOW *newpad(nlines, ncols)
Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**() with a pad as an argument; the routines **prefresh**() or **pnoutrefresh**() should be called instead. Note

that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW *subpad(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**(), which uses screen coordinates, the window is at position (*begin_y*, *begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call **touchwin**() or **touchline**() on *orig* before calling **prefresh**() on *orig*.

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to **wrefresh**() and **wnoutrefresh**() except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Output

These routines are used to manipulate text in windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its

function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **wclrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch**( ) on a position in the window containing a control character will not return the control character, but instead will return one character of the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **winch**( ) and **waddch**( ).) See **wstandout**( ), below.

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch**( ), **mvaddch**( ), and **mvwaddch**( ), are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**( ), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).

Note that *ch* is actually of type **chtype**, not a

character.

Note that **echochar**( ) is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**( ) once for each character in the string.

Note that **addstr**( ), **mvaddstr**( ), and **mvwaddstr**( ) are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend( )
wstandend(win)
standout( )
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**, as well as the macro **COLOR_PAIR**(n). These constants are defined in **<curses.h>** and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**( ). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

wattrset(win, attrs) sets the current attributes of the given window to *attrs*. **wattroff**(win, attrs) turns off the named attributes without turning on or off any other attributes. **wattron**(win, attrs) turns

on the named attributes without affecting any others. **wstandout**(win, attrs) is the same as **wattron**(win, A_STANDOUT). **wstandend**(win, attrs) is the same as **wattrset**(win, **0**), that is, it turns off all attributes.

Note that **wattroff**( ), **wattron**( ), **wattrset**( ), **wstandend**( ), and **wstandout**( ) return **1** at all times.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff**( ), **attron**( ), **attrset**( ), **standend**( ), and **standout**( ) are macros.

beep( )
flash( )

These routines are used to signal the terminal user. **beep**( ) will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash**( ) will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

box(win, vertch, horch)

A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )
werase(win)

These routines copy blanks to every position in the window.

Note that **erase**( ) is a macro.

clear( )
wclear(win)

These routines are like **erase**( ) and **werase**( ), but they also call **clearok**( ), arranging that the screen will be cleared completely on the next call to **wrefresh**( ) for that window, and repainted from scratch.

Note that **clear**( ) is a macro.

clrtobot( )
wclrtobot(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot**( ) is a macro.

clrtoeol( )
wclrtoeol(win)

The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol**( ) is a macro.

delay_output(ms)

Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware ''delete-character'' feature.)

Note that **delch**( ), **mvdelch**( ), and **mvwdelch**( ) are macros.

deleteln( )
wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware ''delete-line'' feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)

The cursor position of the window is placed in the two integer variables *y* and *x*.

Note that **getyx**( ) is a macro, so no "&" is

necessary before the variables *y* and *x*.

getbegyx(win, y, x)
getmaxyx(win, y, x)

The current beginning coordinates (**getbegyx**()) or size (**getmaxyx**()) of the specified window are placed in the two integer variables *y* and *x*.

Note that **getbegyx**() and **getmaxyx**() are macros, so no "&" is necessary before the variables *y* and *x*.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch)

The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch**(), **mvinsch**(), and **mvwinsch**() are macros.

insertln()
winsertln(win)

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln**() is a macro.

move(y, x)
wmove(win, y, x)

The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **wrefresh**() is called. The position specified is relative to the upper left corner of the window, which is (**0**, **0**).

Note that **move**() is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)

These routines overlay text from *srcwin* on top of text from *dstwin* wherever the two windows

overlap. The difference is that **overlay**( ) is non-destructive (blanks are not copied), while **overwrite**( ) is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
This routine provides finer control over the **overlay**( ) and **overwrite**( ) routines. As in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg . . .])
wprintw(win, fmt [, arg . . .])
mvprintw(y, x, fmt [, arg . . .])
mvwprintw(win, y, x, fmt [, arg . . .])
These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)
This routine corresponds to *vfprintf*(3S). It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win) The window is scrolled up one line. This involves moving the lines in the window data structure.

touchwin(win)
touchline(win, start, count)
Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed,

beginning with line *start* .

Input

getch()
wgetch(win)
mvgetch(y, x)
mvwgetch(win, y, x)

A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**(), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**() has been set, the character will also be echoed into the designated window.

When **wgetch**() is called, before getting a character, it will call **wrefresh**() if anything in the window has changed (for example, the cursor has moved or text changed).

When using **getch**(), **wgetch**(), **mvgetch**(), or **mvwgetch**(), do not set both NOCBREAK mode (**nocbreak**()) and ECHO mode (**echo**()) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **wgetch**() encounters a **^D**, it is returned (unlike *stdio* routines, which would return a null string and have a return code of **-1**).

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**() under "Input Options Setting.") Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through,

otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**() below.)

Note that **getch**(), **mvgetch**(), and **mvwgetch**() are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **wgetch**() is made, until a newline, carriage return, or enter key is received. The resulting value (except for this terminating character) is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. See **wgetch**() for how it handles characters differently from *stdio* routines (especially **^D**).

Note that **getstr**(), **mvgetstr**(), and **mvwgetstr**() are macros.

ungetch(c)
Place *c* onto the input queue, to be returned by the next call to **wgetch**().

flushinp()
Throws away any typeahead that has been typed by the user and has not yet been read by the program. Note that **flushinp**() will not throw away any characters supplied by **ungetch**().

inch()
winch(win)
mvinch(y, x)
mvwinch(win, y, x)
The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in **<curses.h>**, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch**(), **winch**(), **mvinch**(), and **mvwinch**() are macros.

scanw(fmt [, arg . . .])
wscanw(win, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**() is called on the window, and the resulting line is used as input for the scan. The return value for these routines is the number of *arg* values that are converted by *fmt*. *arg* values that are not converted are lost. See **wgetstr**() for how it handles strings differently than the *stdio* routines (especially **^D**).

vwscanw(win, fmt, ap) This routine is similar to **vwprintw**() in that it performs a **wscanw**() using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**().

clearok(win, bf) If enabled (*bf* is **TRUE**), the next call to **wrefresh**() with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf) If enabled (*bf* is **TRUE**), *curses* will consider using the hardware ''insert/delete-line'' feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* will very seldom use this feature. (The ''insert/delete-character'' feature is always considered.) This option should be enabled only if your application needs ''insert/delete-line'', for example, for a screen editor. It is disabled by default because ''insert/delete-line'' tends to be visually annoying when used in applications where

it isn't really needed. If "insert/delete-line" cannot be used, *curses* will redraw the changed portions of all lines. Not calling **idlok()** saves approximately 5000 bytes of memory.

leaveok(win, bf)
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
wsetscrreg(win, top, bot)
These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok()** are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok()** is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

Note that **setscrreg()** is a macro.

scrollok(win, bf)
This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is FALSE), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh()** is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok()**.)

Note that **scrollok()** will always return **OK**.

Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin**().

For more information on these options, see the chapter of the *Programmer's Guide* that describes how to write *curses* programs.

cbreak()
nocbreak()

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio*(7)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**() or **nocbreak**() explicitly. Most interactive programs using *curses* will set CBREAK mode.

Note that **cbreak**() performs a subset of the functionality of **raw**(). See **wgetch**() under "Input" for a discussion of how these routines interact with **echo**() and **noecho**().

echo()
noecho()

These routines control whether characters typed by the user are echoed by **wgetch**() as they are typed. Echoing by the tty driver is always disabled, but initially **wgetch**() is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho**(). See **wgetch**() under "Input" for a discussion of how these routines interact with **cbreak**() and **nocbreak**().

nl( )
nonl( )

These routines control whether carriage return is translated into newline on input by **wgetch**( ). Initially, this translation is done; **nonl**( ) turns the translation off. Note that translation by the *tty*(7) driver is disabled in CBREAK mode.

halfdelay(tenths)

Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak**( ) to leave half-delay mode.

intrflush(win, bf)

If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf)

This option enables *curses* to obtain information from the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch**( ) will return a single value representing the function key, as in **KEY_LEFT**. If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit), calling **keypad** (*win*, **TRUE**) will turn it on.

meta(win, bf)

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver (see *termio*(7)). To force 8 bits to be returned, invoke **meta** (*win*, **TRUE**). To force 7 bits to be returned, invoke **meta** (*win*, **FALSE**). The window argument, *win*, is always ignored. If the *terminfo*(4) capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** will be sent to the terminal when **meta** (*win*, **TRUE**) is called and **rmm** will be sent

when **meta** (*win*, FALSE) is called.

nodelay(win, bf)

This option causes **wgetch**() to be a non-blocking call. If no input is ready, **wgetch**() will return **ERR**. If disabled, **wgetch**() will hang until a key is pressed.

notimeout(win, bf)

While interpreting an input escape sequence, **wgetch**() will set a timer while waiting for the next character. If **notimeout**(*win*, **TRUE**) is called, then **wgetch**() will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

raw()
noraw()

The terminal is placed into or out of RAW mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program; however, in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal as they do in CBREAK mode. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.

typeahead(fildes)

curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **wrefresh**() or **doupdate**() is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm**(), or **stdin** in the case that **initscr**() was used, will be used to do this typeahead checking. The **typeahead**() routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **–1,** then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Environment Queries

baudrate() Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar() The user's current erase character is returned.

has_ic() True if the terminal has insert- and delete-character capabilities.

has_il() True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok()** or **idlok()**.

char killchar() The user's current line-kill character is returned.

char *longname() This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr()** or **newterm()**. The area is overwritten by each call to **newterm()** and is not restored by **set_term()**, so the value should be saved between calls to **newterm()** if **longname()** is going to be used with multiple terminals.

Color Manipulation

This section describes the color manipulation routines introduced in this release of *curses*.

start_color()

This routine requires no arguments. It must be called if the user wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after **initscr()**. **start_color()** initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white), and two global variables, **COLORS** and **COLOR_PAIRS** (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the terminal's colors to the values they had when the terminal was just turned on.

init_pair(pair, f, b)

This routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1

and **COLOR_PAIRS-1**. The value of the second and third arguments must be between 0 and **COLORS**-1. If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will be changed to the new definition.

init_color(color, r, g, b)
This routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of the first argument must be between 0 and **COLORS**-1. (See the section COLOR for the default color index.) The last three arguments must each be a value between 0 and 1000. When **init_color**( ) is used, all occurrences of that color on the screen immediately change to the new definition.

has_colors( )
This routine requires no arguments. It returns **TRUE** if the terminal can manipulate colors, **FALSE** otherwise. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

can_change_color( )
This routine requires no arguments. It returns **TRUE** if the terminal supports colors and can change their definitions, **FALSE** otherwise. This routine facilitates writing terminal-independent programs.

color_content(color, &r, &g, &b)
This routine gives users a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of **shorts** for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and **COLORS**-1. The values that will be stored at the addresses pointed to by the last three arguments will be between 0 (no component) and 1000 (maximum amount of component).

pair_content(pair, &f, &b)
This routine allows users to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of **shorts** for storing the foreground and the background color numbers. The value of the first argument must be between 1 and **COLOR_PAIRS-1**. The values that will be stored at the addresses pointed to by the second and

third arguments will be between 0 and **COLORS-1**.

Soft Labels

If desired, *curses* will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *curses* standardizes on 8 labels of 8 characters each. If a *curses* program changes the values of the soft labels, it can restore them only to the default settings for that terminal. Therefore, if before calling a *curses* program a user changes the values of the soft labels, those values cannot be reset when the *curses* program terminates.

slk_init(labfmt)
: In order to use soft labels, this routine must be called before **initscr**() or **newterm**() is called. If **initscr**() winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)
: *labnum* is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A **NULL** string or a **NULL** pointer will put up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh()
slk_noutrefresh() These routines correspond to the routines **wrefresh()** and **wnoutrefresh()**. Most applications would use **slk_noutrefresh()** because a **wrefresh()** will most likely soon follow.

char *slk_label(labnum)
The current label for label number *labnum* is returned, in the same format as it was in when it was passed to **slk_set()**; that is, how it looked prior to being justified according to the *labfmt* argument of **slk_set()**.

slk_clear() The soft labels are cleared from the screen.

slk_restore() The soft labels are restored to the screen after a **slk_clear()**.

slk_touch() All of the soft labels are forced to be output the next time a **slk_noutrefresh()** is performed.

slk_attron(attrs)
slk_attrset(attrs)
slk_attroff(attrs) These routines correspond to **attron()**, **attrset()**, and **attroff()**. They will have effect only if soft labels are simulated on the bottom line of the screen.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

def_prog_mode()
def_shell_mode() Save the current terminal modes as the ''program'' (in **curses**) or ''shell'' (not in **curses**) state for use by the **reset_prog_mode()** and **reset_shell_mode()** routines. This is done automatically by **initscr()**.

reset_prog_mode()
reset_shell_mode() Restore the terminal to ''program'' (in **curses**) or ''shell'' (out of *curses*) state. These are done automatically by **endwin()** and **doupdate()** after an **endwin()**, so they normally would not be called.

resetty( )
savetty( )

These routines save and restore the state of the terminal modes. **savetty**( ) saves the current state of the terminal in a buffer and **resetty**( ) restores the state to what it was at the last call to **savetty**( ).

getsyx(y, x)

The current coordinates of the virtual screen cursor are returned in *y* and *x*. If **leaveok**( ) is currently **TRUE**, then **−1**,**−1** will be returned. If lines have been removed from the top of the screen using **ripoffline**( ), *y* and *x* include these lines; therefore, *y* and *x* should be used only as arguments for **setsyx**( ).

Note that **getsyx**( ) is a macro, so no "&" is necessary before the variables *y* and *x*.

setsyx(y, x)

The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **−1**, then **leaveok**( ) will be set. The two routines **getsyx**( ) and **setsyx**( ) are designed to be used by a library routine which manipulates *curses* windows but does not want to change the current position of the program's cursor. The library routine would call **getsyx**( ) at the beginning, do its manipulation of its own windows, do a **wnoutrefresh**( ) on its windows, call **setsyx**( ), and then call **doupdate**( ).

ripoffline(line, init)

This routine provides access to the same facility that **slk_init**( ) uses to reduce the size of the screen. **ripoffline**( ) must be called before **initscr**( ) or **newterm**( ) is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr**( ), the routine *init*( ) is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine.

ripoffline( ) can be called up to five times before

calling **initscr**() or **newterm**().

scr_dump(filename) The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename) The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**(). **ERR** is returned if the contents of *filename* are not compatible with the current release of *curses* software. The next call to **doupdate**() will restore the screen to what it looked like in the dump file.

scr_init(filename) The contents of *filename* are read in and used to initialize the *curses* data structures about what the terminal currently has on its screen. If the data is determined to be valid, *curses* will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**() would be used after **initscr**() or a *system*(3S) call to share the screen with another process which has done a **scr_dump**() after its **endwin**() call. The data will be declared invalid if the *terminfo*(4) capability **nrrmc** is true or the time-stamp of the tty is old. Note that **keypad**(), **meta**(), **slk_clear**(), **curs_set**(), **flash**(), and **beep**() do not affect the contents of the screen, but will make the tty's time-stamp old.

curs_set(visibility) The cursor state is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, **ERR** is returned.

draino(ms) Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)
This routine indicates to *curses* that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire

screen would be subject to even more communication noise. Just redrawing the single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms) Sleep for *ms* milliseconds.

mvcur(oldrow, oldcol, newrow, newcol)
Low-level cursor motion.

Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** (see *terminfo*(4)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer (see *layers*(1)), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files **<curses.h>** and **<term.h>** should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings (including the output of **tparm**( )) should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines (see **endwin**( )).

setupterm(term, fildes, errret)
Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is NULL, the environment variable **TERM** will be used. All output is to the file descriptor *fildes*. If

errret is not NULL, then **setupterm**() will return **OK** or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **−1** means that the *terminfo*(4) database could not be found. If *errret* is NULL, **setupterm**() will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**() returns successfully, the variable **cur_term** (of type **TERMINAL ***) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**() again. Further calls to **setupterm**() will allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)
nterm is of type **TERMINAL ***. **set_curterm**() sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)
oterm is of type **TERMINAL ***. **del_curterm**() frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**() has been called.

restartterm(term, fildes, errret)
Similar to **setupterm**(), except that it is called after restoring memory to a previous state; for example, after a call to **scr_restore**(). It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

char *tparm(str, p_1, p_2, ..., p_9)
Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc) Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from **tparm**(), **tgetstr**(), **tigetstr**() or **tgoto**(). *count* is the number of lines affected, or **1** if not applicable. *putc* is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str) A routine that calls **tputs** (*str*, **1**, **putchar**).

vidputs(attrs, putc) Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*().

vidattr(attrs) Similar to **vidputs**(), except that it outputs through *putchar*(3S).

The following routines return the value of the capability corresponding to the character string containing the *terminfo*(4) *capname* passed to them. For example, **rc** = **tigetstr("acsc")** causes the value of **acsc** to be returned **in rc.**

tigetflag(capname) The value **–1** is returned if *capname* is not a boolean capability. The value **0** is returned if *capname* is not defined for this terminal.

tigetnum(capname) The value **–2** is returned if *capname* is not a numeric capability. The value **–1** is returned if *capname* is not defined for this terminal.

tigetstr(capname) The value (char *) **–1** is returned if *capname* is not a string capability. A null value is returned if *capname* is not defined for this terminal.

char *boolnames[ ], *boolcodes[ ], *boolfnames[ ]
char *numnames[ ], *numcodes[ ], *numfnames[ ]
char *strnames[ ], *strcodes[ ], *strfnames[ ]
These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name) Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename) Get the boolean entry for *codename*.

tgetnum(codename) Get numeric entry for *codename*.

char *tgetstr(codename, area)
Return the string entry for *codename*. If *area* is not NULL, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**() should be used to output the returned string.

char *tgoto(cap, col, row)
Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**().

tputs(str, affcnt, putc) See **tputs**() above, under "Terminfo-Level Manipulations".

Miscellaneous

traceoff()
traceon() Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license.

unctrl(c) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

unctrl() is a macro, defined in **<unctrl.h>**, which is automatically included by **<curses.h>**.

char *keyname(c) A character string corresponding to the key *c* is returned.

filter() This routine is one of the few that is to be called before **initscr**() or **newterm**() is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**() is **curscr**, the next call to **wrefresh**() with any window will cause the screen to be cleared and repainted from scratch. If

the window argument to **wrefresh**() is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**(), **overwrite**(), and **copywin**() may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode()	Replaced by **cbreak**().
fixterm()	Replaced by **reset_prog_mode**().
gettmode()	A no-op.
nocrmode()	Replaced by **nocbreak**().
resetterm()	Replaced by **reset_shell_mode**().
saveterm()	Replaced by **def_prog_mode**().
setterm()	Replaced by **setupterm**().

ATTRIBUTES

The following video attributes, defined in <**curses.h**>, can be passed to the routines **wattron**(), **wattroff**(), and **wattrset**(), or OR'ed with the characters passed to **waddch**().

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract character (described under **winch**())
A_ATTRIBUTES	Bit-mask to extract attributes (described under **winch**())
A_NORMAL	Bit mask to reset all attributes off (for example: **wattrset** (win, A_NORMAL)
COLOR_PAIR(n)	Color-pair defined in n (note that this is a macro)

The following bit-masks may be AND'ed with characters returned by **winch**().

A_CHARTEXT	Extract character
A_ATTRIBUTES	Extract attributes

A_COLOR	Extract color-pair field information

The following macro is the reverse of COLOR_PAIR(n).

PAIR_NUMBER(attrs)	Returns the pair number associated with the COLOR_PAIR(n) attribute (note that this is a macro)

COLORS

In <**curses.h**> the following macros are defined to have the numeric value shown. These are the default colors. *curses* also assumes that color 0 (zero) is the default background color for all terminals.

COLOR_BLACK	0
COLOR_BLUE	1
COLOR_GREEN	2
COLOR_CYAN	3
COLOR_RED	4
COLOR_MAGENTA	5
COLOR_YELLOW	6
COLOR_WHITE	7

FUNCTION KEYS

The following function keys, defined in <**curses.h**>, might be returned by **wgetch**() if **keypad**() has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserve(
KEY_F(n)	(KEY_F0+(n))	Formula for f_n.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode

KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)
		keypad is arranged like this:
		A1 up A3
		left B2 right
		C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key
KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key
KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key
KEY_MARK	0554	mark key
KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key

KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key
KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key
KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key
KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch**(). When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character

listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	−	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	−	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

DIAGNOSTICS

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **getsyx()**, **getyx()**, **getbegyx()**, **getmaxyx()**. For these macros, no useful value is returned.

Routines that return pointers always return **(type *)** NULL on error.

BUGS

Currently typeahead checking is done using a nodelay read followed by an **ungetch()** of any character that may have been read. Typeahead checking is done only if **wgetch()** has been called at least once. This may change when proper kernel support is available. Programs which use a mixture of

their own input routines with *curses* input routines may wish to call **typeahead(−1)** to turn off typeahead checking.

The argument to **napms**( ) is currently rounded up to the nearest second.

draino (ms) only works for *ms* equal to **0**.

WARNINGS

To use the new *curses* features, use the Release 3.2 version of *curses* on UNIX System V Release 3.2. All programs that ran with Release 2, Release 3.0 or Release 3.1 *curses* will also run on UNIX System V Release 3.2. You can link applications with object files based on Release 2 or Release 3.0 *curses/terminfo* with both the Release 3.1 and Release 3.2 *libcurses.a* library; however, you cannot link applications with object files based on Release 3.1 or Release 3.2 *curses/terminfo* with the Release 2 or Release 3.0 *libcurses.a* library.

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names **erase**( ) and **move**( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or **#undef move**( ) and **erase**( ) in the *plot*(3X) code.

Between the time a call to **initscr**( ) and **endwin**( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

If a pointer passed to a routine as a window argument is null or out of range, the results are undefined (core may be dumped).

Several *curses* routines such as **setupterm**( ) and **resetterm**( ) make TCSETA *ioctl*(2) calls on the terminal. This means that programs that use *curses* will typically get suspended when executed in background, even if ''stty -tostop'' has been done. Refer to *csh*(1), *stty*(1) and *termio*(7) for further information.

SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5).
termio(7), tty(7) in the *System Administrator's Reference Manual.*
curses/terminfo chapter of the *Programmer's Guide*.

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (char *s);

DESCRIPTION

cuserid generates a character-string representation of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (**\0**) will be placed at *s[0]*.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
#include <dbm.h>

typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

Note: the dbm library has been superceded by ndbm(3B), and is now implemented using ndbm. These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

for (key = firstkey(); key.dptr != NULL; key = nextkey(key))

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

SEE ALSO

ndbm(3B)

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (CALL call);

void undial (int fd);

DESCRIPTION

dial returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <*dial.h*> header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the <*dial.h*> header file is:

```
typedef struct {
        struct termio *attr;    /* pointer to termio attribute struct */
        int           baud;     /* transmission data rate */
        int           speed;    /* 212A modem: low=300, high=1200 */
        char          *line;    /* device name for out-going line */
        char          *telno;   /* pointer to tel-no digits string */
        int           modem;    /* specify modem control for direct lines */
        char          *device;  /*Will hold the name of the device used
                                  to make a connection */
        int           dev_len;  /* The length of the device used to make
                                  connection */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if **speed** set to 1200 **baud** must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L-devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L-devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev_len* is the length of the device name that is copied into the array device.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

alarm(2), read(2), write(2).
termio(7) in the *System Administrator's Reference Manual*.
uucp(1C) in the *User's Reference Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the <*dial.h*> header file.

INTRPT	−1	/* interrupt occurred */
D_HUNG	−2	/* dialer hung (no return from write) */
NO_ANS	−3	/* no answer within 10 seconds */
ILL_BD	−4	/* illegal baud-rate */
A_PROB	−5	/* acu problem (open() failure) */
L_PROB	−6	/* line problem (open() failure) */
NO_Ldv	−7	/* can't open LDEVS file */
DV_NT_A	−8	/* requested device not available */
DV_NT_K	−9	/* requested device not known */
NO_BD_A	−10	/* no device available at requested baud */
NO_BD_K	−11	/* no device known at requested baud */

WARNINGS

The *dial* (3C) library function is not compatible with Basic Networking Utilities on UNIX System V Release 2.0.

Including the **<dial.h>** header file automatically includes the **<termio.h>** header file.

The above routine uses <stdio.h>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of ''touching'' the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

NAME

difftime – compute difference between two calendar times

SYNOPSIS

#include <time.h>

double difftime (time_t time1, time_t time0);

DESCRIPTION

difftime computes the difference between two calendar times: **time1 - time0.**

SEE ALSO

mktime(3C), time(2).

DIAGNOSTICS

difftime returns the difference expressed in seconds as a double.

NAME

directory: opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations (System V)

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

DIR *opendir (const char *filename);

struct dirent *readdir (DIR *dirp);

long telldir (DIR *dirp);

void seekdir (DIR *dirp, long loc);

void rewinddir (DIR *dirp);

int closedir (DIR *dirp);

DESCRIPTION

The inclusion of *<dirent.h>* selects the System V versions of these routines. For the 4.3BSD versions, include *<sys/dir.h>*.

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot *malloc* enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory.

telldir returns the current location associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation from which *loc* was obtained was performed. Values returned by *telldir* are good only if the directory has not changed due to compaction or expansion. This is not a problem with System V, but it may be with some file system types.

rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir closes the named *directory stream* and frees the **DIR** structure. It returns a value of zero if successful. Otherwise, it returns –1 indicating an error.

DIAGNOSTICS

If an error occurs as a result of an operation, the global variable *errno* is set to one of the following values:

opendir:

[ENOTDIR] A component of *filename* is not a directory.

[EACCES] A component of *filename* denies search permission.

[ENAMETOOLONG]
The length of the *filename* argument exceeds {**PATH_MAX**}.

[ENOENT] The named directory does not exist.

[EMFILE] The maximum number of file descriptors are currently open.

[EFAULT] *Filename* points outside the allocated address space.

[EAGAIN] A call to *malloc* failed.

[ENOMEM] A call to *malloc* failed.

readdir:

[ENOENT] The current file pointer for the directory is not located at a valid entry.

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

telldir, *seekdir*, and *closedir:*

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the **DIR** stream has been closed.

EXAMPLE

Sample code that searches a directory for entry *name*:

```
dirp = opendir(".");
if (dirp == NULL) {
        return NOT_FOUND;
}
while ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
                (void) closedir(dirp);
                return FOUND;
```

```
                }
        }
        (void) closedir(dirp);
        return NOT_FOUND;
```

SEE ALSO

getdents(2), malloc(3C), malloc(3X), scandir(3C), dirent(4), directory(3B).

WARNINGS

rewinddir is implemented as a macro, so its function address cannot be taken.

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir, dirfd – directory operations (4.3BSD)

SYNOPSIS

#include <sys/types.h>
#include <sys/dir.h>

DIR *opendir(char *filename);

struct direct *readdir(DIR *dirp);

long telldir(DIR *dirp);

void seekdir(DIR *dirp, long loc);

void rewinddir(DIR *dirp);

void closedir(DIR *dirp);

int dirfd(DIR *dirp)

DESCRIPTION

The inclusion of *<sys/dir.h>* selects the 4.3BSD versions of these routines. For the System V versions, include *<dirent.h>*.

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer **NULL** is returned if *filename* cannot be accessed, or if it cannot *malloc*(3) enough memory to hold the whole thing.

readdir returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory or detecting an invalid *seekdir* operation.

telldir returns the current location associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed. Values returned by *telldir* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir.*

rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir closes the named *directory stream* and frees the structure associated with the DIR pointer.

dirfd returns the integer file descriptor associated with the named *directory stream*, see open(2).

Sample code that searches a directory for entry ''name'':

```
len = strlen(name);
dirp = opendir(".");
if (dirp == NULL) {
    return NOT_FOUND;
}
while ((dp = readdir(dirp)) != NULL) {
    if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
        closedir(dirp);
        return FOUND;
    }
}
closedir(dirp);
return NOT_FOUND;
```

SEE ALSO

open(2), close(2), read(2), lseek(2), directory(3C)

NAME

disassembler – disassemble a MIPS instruction and print the results

SYNOPSIS

```
int disassembler (iadr, regstyle, get_symname, get_regvalue,
                  get_bytes, print_header)
unsigned          iadr;
int               regstyle;
char              *(*get_symname)();
int               (*get_regvalue)();
long              (*get_bytes)();
void              (*print_header)();
```

DESCRIPTION

Disassembler disassembles and prints a MIPS machine instruction on *stdout.*

Iadr is the instruction address to be disassembled. *Regstyle* specifies how registers are named in the disassembly; if the value is 0, compiler names are used; otherwise, hardware names are used.

The next four arguments are function pointers, most of which give the caller some flexibility in the appearance of the disassembly. The only function that **MUST** be provided is *get_bytes.* All other functions are optional. *Get_bytes* is called with no arguments and returns the next byte(s) to disassemble.

Get_symname is passed an address, which is the target of a *jal* instruction. If **NULL** is returned or if *get_symname* is **NULL**, the *disassembler* prints the address; otherwise, the string name is printed as returned from *get_symname.* If *get_regvalue* is not **NULL**, it is passed a register number and returns the current contents of the specified register. *Disassembler* prints this information along with the instruction disassembly. If *print_header* is not **NULL**, it is passed the instruction address *iadr* and the current instruction to be disassembled, which is the return value from *get_bytes. Print_header* can use these parameters to print any desired information before the actual instruction disassembly is printed.

If *get_bytes* is **NULL**, the *disassembler* returns -1 and errno is set to **EINVAL;** otherwise, the number of bytes that were disassembled is returned. If the disassembled word is a jump or branch instruction, the instruction in the delay slot is also disassembled.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldfcn(4).

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

#include <math.h>

double drand48 (void);

double erand48 (unsigned short xsubi[3]);

long lrand48 (void);

long nrand48 (unsigned short xsubi[3]);

long mrand48 (void);

long jrand48 (unsigned short xsubi[3]);

void srand48 (long seedval);

unsigned short *seed48 (unsigned short seed16v[3]);

void lcong48 (unsigned short param[7]);

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$$a = \text{5DEECE66D}_{16} = 273673163155_8$$
$$c = \text{B}_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (left-most) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer, and must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value 330E_{16}.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*, and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

SEE ALSO

rand(3C).

NAME

dsopen, dsclose – communicate with generic SCSI devices

SYNOPSIS

#include <dslib.h>

struct dsreq *dsopen(opath, oflags)

dsclose(dsp)

testunitready00(dsp)
requestsense03(dsp, data, datalen, vu)
read08(dsp, data, datalen, lba, vu)
write0a(dsp, data, datalen, lba, vu)
inquiry12(dsp, data, datalen, vu)
modeselect15(dsp, data, datalen, save, vu)
reserveunit16(dsp, data, datalen, tpr, tpdid, extent, resid, vu)
releaseunit17(dsp, tpr, tpdid, extent, resid, vu)
modesense1a(dsp, data, datalen, pagectrl, pagecode, vu)
senddiagnostic1d(dsp, data, datalen, self, dofl, uofl, vu)

readcapacity25(dsp, data, datalen, lba, pmi, vu)
readextended28(dsp, data, datalen, lba, vu)
writeextended2a(dsp, data, datalen, lba, vu)

getfd(dsp)
doscsireq(fd, dsp)
void fillg0cmd(dsp, cmdbuf, b0, ..., b5)
void fillg1cmd(dsp, cmdbuf, b0, ..., b9)
void filldsreq(dsp, data, datalen, flags)
void vtostr(value, table)

extern int dsdebug;
extern long dsreqflags;
DSDBG(statement; ...)

```
struct dsreq *dsp;
struct vtab *table;
char *opath, *cmdbuf, *data;
char b0, ..., b9, dofl, extent, pagecode, pagectrl, pmi, resid,
          save, self, tpdid, tpr, uofl, vu;
int fd, oflags;
long datalen, lba, value;
```

DESCRIPTION

These routines form the basis for a simplified interface to *ds*(7m) devices. They are included in a program by compiling with the *-lds* option. An application would typically use *dsopen*, *dsclose*, and a set of command-specific routines such as *testunitready00*. The source to this library may be obtained by loading the *std.sw.giftssrc* image, with the source code for the library in the files *dstab.c* and *dslib.c* in the directory */usr/people/4Dgifts/examples/devices*.

The number of truly general SCSI commands is quite limited, so provision is made for supporting vendor-specific commands. A set of helper routines (*fillg0cmd*, etc.) is used as the basis for this. *testunitready00*, for instance, is implemented as:

```
testunitready00(dsp)
struct dsreq *dsp;
{
  fillg0cmd(dsp, CMDBUF(dsp), G0_TEST, 0, 0, 0, 0, 0);
  filldsreq(dsp, 0, 0, DSRQ_READ|DSRQ_SENSE);
  return(doscsireq(getfd(dsp), dsp));
}
```

Note that many of these routines depend upon the exact setup of the *dsreq* structure used by *dsopen*. It is therefore *not* recommended that users attempt to use independently derived *dsreq* structures with them.

dsopen passes *opath* and *oflags* to the *open* system call. If the *open* succeeds, *dsopen* allocates and fills a *dsreq* structure, along with some associated context information. *dsclose* deallocates the specified *dsreq* structure, then calls *close* to close the device.

fillg0cmd and *fillg1cmd* are used to fill Group 0 and 1 command buffers, respectively. *filldsreq* fills a *dsreq* structure with commonly needed data. The value of dsreqflags is ORed into the *ds_flags* field. This is useful if you want a flag (such as DSRQ_SENSE) set for some or all commands, as it allows you to avoid duplicating the library routines when you need a special flag set. *doscsireq* issues the SCSI *ioctl*, performs a variety of error-handling functions, and returns the SCSI status byte.

ds_vtostr Takes a value, and a table to look it up in. If the value is found in the given table, a string describing the value is returned, else the empty string. Five tables are provided: *dsrqnametab* for the DSRQ_* flags, *dsrtnametab* for the DSRT_* flags, *cmdstatustab* for the SCSI status byte return in ds_status, *msgnametab* for the SCSI message bytes, and *cmdnametab* for the SCSI commands, such as Testunitready (value is the command byte; G0_TEST in this case).

The *dsdebug* variable, and the DSDBG() macro can be used to enable debug printfs, and to add your own. If the *dsdebug* is non-zero, debugging information is printed by the library routines. The *DSDBG* macro is used for this purpose. A more or less arbitrary sequence of statements may be used within the parentheses of the DSDBG macro, but some form of print statement is most frequently used.

Overlay structures define the layouts of the three (Group 0, 1, 6) Common Command Set command buffers. Bytes are named both by position (g0_b0) and by typical function in the command buffer (g1_op_code).

Mnemonic names are also defined for all CCS command codes (G0_TEST), message bytes (MSG_ABORT), and status bytes (STA_BUSY). There are also a number of macros suitable for accessing *dsreq* structures, SCSI byte and bit fields, etc.

A set of structures contains values, name strings, and descriptions for commonly used codes and values. The structures document DSRQ_* and DSRT_* codes, CCS command codes, and CCS status and message bytes. They are principally useful in generating explicit error messages.

EXAMPLE PROGRAM

The following code fragment illustrates simple use of the library, and of some /dev/scsi support macros. If you have installed the *std.sw.giftssrc* image, the full source code for this program may be found in the file */usr/people/4Dgifts/examples/devices/inquire.c*,

```
while (--argc > 0) {
     fn = *++argv;
     printf("%-17s  ", fn);
     if ((dsp = dsopen(fn, O_RDONLY)) == NULL) {
          fflush(stdout);
          perror("cannot open");
          continue;
     }

     if(inquiry12(dsp, inqbuf, sizeof inqbuf, 0) != 0)
          printf("%-10s inquiry failure0, "---");
     else {
          pdt = DATABUF(dsp)[0] & 0x7F;
          if (DATASENT(dsp) >= 1)
               printf("%-10s", pdt_types[(pdt<NPDT) ? pdt : NPDT-1]);
          if (DATASENT(dsp) >= 16) printf("  %-12.8s", &DATABUF(dsp)[8]);
          if (DATASENT(dsp) >= 32) printf("  %.16s",   &DATABUF(dsp)[16]);
          if (DATASENT(dsp) >= 36) printf("  %.4s",    &DATABUF(dsp)[32]);
          /*  do test unit ready only if inquiry successful, since many
               devices, such as tapes, return inquiry info, even if
               not ready (i.e., no tape in a tape drive). */
```

```
          if(testunitready00(dsp) != 0) {
              printf("  %s0,
                  (RET(dsp)==DSRT_NOSEL) ? "cannot select" : "not ready"
          }
          else
              printf("  ready0);
      }
      dsclose(dsp);
  }
```

Each device is opened, and the necessary data structures created. An inquiry is done to see if the device exists; if so, it's type is printed. A test unit ready is done to see if the device is ready for i/o. Finally, the device is closed, releasing the data structures. The normal output is of the form:

/dev/scsi/sc0d2l0 Tape ARCHIVE VIPER 150 21247 -605 not ready

DIAGNOSTICS

dsopen returns a NULL pointer on failure. *doscsireq* returns -1 on absolute failure, and the status byte otherwise. A status byte of 0xff indicates an invalid status byte because the scsi command didn't complete. The RET(dsp) macro returns a result code, which may be consulted for any error or 'unusual' status from the driver.

SEE ALSO

ds(7m)

NAME

dup2 – duplicate an open file descriptor

SYNOPSIS

#include <unistd.h>

int dup2 (int fildes, int fildes2);

DESCRIPTION

Fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than NOFILES. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

dup2 will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] The maximum number of file descriptors are currently open (see *getdtablesize*(3)).

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

SYNOPSIS

char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;

DESCRIPTION

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

emulate_branch – MIPS branch emulation

SYNOPSIS

#include <signal.h>

emulate_branch(scp, branch_instruction)
struct sigcontext *scp;
unsigned long branch_instruction;

DESCRIPTION

Emulate_branch is passed a signal context structure and a branch instruction. It emulates the branch based on the register values in the signal context structure. It modifies the value of the program counter in the signal context structure (*sc_pc*) to the target of the branch instruction. The program counter must initially be pointing at the branch and the register values must be those at the time of the branch. If the branch is not taken the program counter is advanced to point to the instruction after the delay slot (*sc_pc* += 8).

In the case the branch instruction is a *branch on coprocessor 2 or 3* instruction *emulate_branch* can't emulate or execute the branch currently.

Programs using this function must load */usr/lib/fixade.o*.

RETURN VALUE

Emulate_branch returns a 0 if the branch was emulated successfully. An non-zero value indicates the value passed as a branch instruction was not a branch instruction.

FILES

/usr/lib/fixade.o

SEE ALSO

signal(2), sigset(2)

NAME

end, etext, edata, eprol, _ftext, _fdata, _fbss, _procedure_table, _procedure_table_size, _procedure_string_table, _gp – loader defined symbols in a program

SYNOPSIS

extern end;
extern etext;
extern edata;
extern eprol;
extern _ftext;
extern _fdata;
extern _fbss;
extern _procedure_table;
extern _procedure_table_size;
extern _procedure_string_table;
extern _gp;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents except for *_procedure_table* and *_procedure_string_table.*

The first three of these are standard UNIX linker symbols. The others are MIPS specific.

Except for *eprol* these are all names of loader defined symbols.

The address of *etext* is the first address above the program text, *edata* is the first address above the initialized data region, and *end* is the first address above the uninitialized data region.

The address of *_ftext* is the first address in the program text, *_fdata* is the first address in the initialized data region, and *_fbss* is the first address in the uninitialized data region.

eprol is only defined if a profiling runtime startup is used (see *prof*(1)). If present, *eprol* is the address of the first instruction after normal initializations but before profiling is started up (that is, before *monstartup*(3C) is called) and before shared-library jump table initializations are done.

_gp is the address of the region of global data accessed by offsets of the global-pointer register (ie, its address is the run-time value of the global-pointer register.)

When execution begins, the program break coincides with *_end,* but it is reset by the routines *brk*(2), *malloc*(3), standard input/output (*stdio*(3)), the profile (**–p**) option of *cc*(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see *brk*(2).

The loader defined symbols *_procedure_table*, *_procedure_table_size*, and *_procedure_string_table* refer to data structures of the runtime procedure table. Since these are loader defined symbols the data structures are built by *ld(1)* only if they are referenced. See the include file <*sym.h*> for the definition of the runtime procedure table and see the include file <*exception.h*> for its uses.

SEE ALSO

brk(2), malloc(3c)

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (double x);

double erfc (double x);

DESCRIPTION

erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

erfc, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g., for $x = 10$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, expm1, log, log10, log1p, pow, fexp, fexpm1, flog, flog10, flog1p – exponential, logarithm, power

SYNOPSIS

#include <math.h>

double exp(double x);

float fexp(float x);

double expm1(double x);

float fexpm1(float x);

double log(double x);

float flog(float x);

double log10(double x);

float flog10(float x);

double log1p(double x);

float flog1p(float x);

double pow(double x, double y);

DESCRIPTION

exp and *fexp* return the exponential function of x for double and float data types respectively.

expm1 and *fexpm1* return exp(x)–1 accurately even for tiny x for double and float data types respectively.

log and *flog* return the natural logarithm of x for double and float data types respectively.

log10 and *flog10* return the logarithm of x to base 10 for double and float data types respectively.

log1p and *flog1p* return log(1+x) accurately even for tiny x for double and float data types respectively.

pow(x,y) returns x^y.

ERROR (due to Roundoff etc.)

exp(x), *log(x)*, *expm1(x)* and *log1p(x)* are accurate to within an *ulp*, and *log10(x)* to within about 2 *ulps*; an *ulp* is one *U*nit in the *L*ast *P*lace. The error in *pow(x,y)* is below about 2 *ulps* when its magnitude is moderate, but increases as *pow(x,y)* approaches the over/underflow thresholds until almost as many bits could be lost as are occupied by the floating–point format's exponent field; that is, 11 bits for IEEE 754 Double. No such drastic loss has been exposed by testing; the worst errors observed have been below

300 *ulps* for IEEE 754 Double. Moderate values of pow are accurate enough that pow(integer,integer) is exact until it is bigger than 2**53 for IEEE 754.

DIAGNOSTICS

exp returns infinity when the correct value would overflow, or the smallest non-zero value when the correct value would underflow.

log and *log10* returns the default quiet NaN when *x* is less than zero indicating the invalid operation. *log* and *log10* return -Infinity when *x* is zero.

pow returns Infinity when *x* is zero and *y* is non-positive. *pow* returns NaN when *x* is negative and *y* is not an integer, indicating the invalid operation. When the correct value for *pow* would overflow or underflow *pow* returns +Infinity or zero, respectively.

NOTES

Pow(x,0) returns x**0 = 1 for all x including x = 0, ∞ and NaN. Previous implementations of pow may have defined x**0 to be undefined in some or all of these cases. Here are reasons for returning x**0 = 1 always:

(1) Any program that already tests whether x is zero (or infinite or *NaN*) before computing x**0 cannot care whether 0**0 = 1 or not. Any program that depends upon 0**0 to be invalid is dubious anyway since that expression's meaning and, if invalid, its consequences vary from one computer system to another.

(2) Some Algebra texts (e.g. Sigler's) define x**0 = 1 for all x, including x = 0. This is compatible with the convention that accepts a[0] as the value of polynomial

p(x) = a[0]*x**0 + a[1]*x**1 + a[2]*x**2 +...+ a[n]*x**n

at x = 0 rather than reject a[0]*0**0 as invalid.

(3) Analysts will accept 0**0 = 1 despite that x**y can approach anything or nothing as x and y approach 0 independently. The reason for setting 0**0 = 1 anyway is this:

If x(z) and y(z) are *any* functions analytic (expandable in power series) in z around z = 0, and if there x(0) = y(0) = 0, then x(z)**y(z) → 1 as z → 0.

(4) If 0**0 = 1, then ∞**0 = 1/0**0 = 1 too; and then *NaN***0 = 1 too because x**0 = 1 for all finite and infinite x, i.e., independently of x.

SEE ALSO

math(3M)

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (FILE *stream);

int fflush (FILE *stream);

DESCRIPTION

fclose causes the named *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are written to the file; any unread buffered data are discarded. The stream is disassociated from the file.

fclose is performed automatically for all open files upon calling *exit*(2).

fflush causes any unwritten buffered data for the named *stream* to be written to that file. The *stream* remains open.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

#include <stdio.h>

int ferror (FILE *stream);

int feof (FILE *stream);

void clearerr (FILE *stream);

int fileno (FILE *stream);

DESCRIPTION

ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

The following symbolic values in *<unistd.h>* define the file descriptors that are associated with the C language *stdin*, *stdout*, and *stderr* when an application is started:

Name	Description	Value
STDIN_FILENO	Standard input value, *stdin*	0
STDOUT_FILENO	Standard output value, *stdout*	1
STDERR_FILENO	Standard error value, *stderr*	2

NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S), stdio(3S).

NAME

flock – apply or remove an advisory lock on an open file

SYNOPSIS

#include <sys/file.h>

int flock(int fd, int operation);

DESCRIPTION

Flock applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is the inclusive or of **LOCK_SH** or **LOCK_EX** and, possibly, **LOCK_NB**. To unlock an existing lock, *operation* should be **LOCK_UN**.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e., processes may still access files without using advisory locks possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If **LOCK_NB** is included in *operation*, then this will not happen; instead the call will fail and the error **EWOULDBLOCK** will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through *dup*(3C) or *fork*(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

In C++, the function name *flock* collides with the structure declaration *flock* (which is defined in *<sys/fcntl.h>* and included in *<sys/file.h>*). To get around this, when using *flock*() in C++, one must define **_BSD_COMPAT** before including *sys/file.h*

RETURN VALUE

Zero is returned if the operation was successful; on an error a −1 is returned and an error code is left in the global location *errno*.

ERRORS

The *flock* call fails if:

[EWOULDBLOCK]	The file is locked and the **LOCK_NB** option was specified.
[EBADF]	The argument *fd* is an invalid descriptor.
[EINVAL]	The argument *fd* refers to an object other than a file.

SEE ALSO

open(2), close(2), dup(3C), execve(2), fcntl(2), fork(2), lockf(3)

BUGS

Unlike BSD *flock*(2), attempts to acquire an exclusive lock on an *fd* opened for reading but not writing will fail with **EBADF**, as will attempts to acquire a shared lock on a write-only *fd*.

NAME

floor, ffloor, ceil, fceil, fmod, fabs, rint, trunc, ftrunc – floor, ceiling, remainder, absolute value, nearest integer, and truncation functions

SYNOPSIS

#include <math.h>

double floor (double x);

float ffloor (float x);

double ceil (double x);

float fceil (float x);

double trunc (double x);

float ftrunc (float x);

double fmod (double x, double y);

double fabs (double x);

double rint(double x);

DESCRIPTION

floor and *ffloor* return the largest integer not greater than x for double and float data types respectively.

ceil and *fceil* return the smallest integer not less than x for double and float data types respectively.

trunc and *ftrunc* return the integer (represented as a floating-point number) of x with the fractional bits truncated for double and float data types respectively.

fmod returns the floating-point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

fabs returns the absolute value of x, $|x|$.

rint returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

NOTES

In the default rounding mode, to nearest, *rint(x)* is the integer nearest x with the additional stipulation that if $|\text{rint}(x)-x|=1/2$ then *rint(x)* is even. Other rounding modes can make *rint* act like *floor*, or like *ceil*, or round towards zero.

Another way to obtain an integer near x is to declare (in C)

double x;   int k;   k = x;

The MIPS C compilers round *x* towards zero to get the integer *k*. Also note that, if *x* is larger than *k* can accommodate, the value of *k* and the presence or absence of an integer overflow are hard to detect.

The routine *fabs* is in libc.a rather than libm.a.

SEE ALSO

abs(3C), ieee(3M)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen (const char *filename, const char *type);

FILE *freopen (const char *filename, const char *type,
FILE *stream);

FILE *fdopen (int fildes, const char *type);

DESCRIPTION

fopen opens the file named by *filename* and associates a *stream* with it. *fopen* returns a pointer to the FILE structure associated with the *stream*.

Filename points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin, stdout** and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), stdio(3S).

DIAGNOSTICS

fopen, *fdopen*, and *freopen* return a NULL pointer on failure.

NAME

fp_class – classes of IEEE floating-point values

SYNOPSIS

```
#include <fp_class.h>

int fp_class_d(double x);

int fp_class_f(float x);
```

DESCRIPTION

These routines are used to determine the class of IEEE floating-point values. They return one of the constants in the file *<fp_class.h>* and never cause an exception even for signaling NaN's. These routines are to implement the recommended function class(*x*) in the appendix of the IEEE 754-1985 standard for binary floating-point arithmetic.

The constants in *<fp_class.h>* refer to the following classes of values:

Constant	Class
FP_SNAN	Signaling NaN (Not-a-Number)
FP_QNAN	Quiet NaN (Not-a-Number)
FP_POS_INF	$+\infty$ (positive infinity)
FP_NEG_INF	$-\infty$ (negative infinity)
FP_POS_NORM	positive normalized non-zero
FP_NEG_NORM	negative normalized non-zero
FP_POS_DENORM	positive denormalized
FP_NEG_DENORM	negative denormalized
FP_POS_ZERO	+0.0 (positive zero)
FP_NEG_ZERO	–0.0 (negative zero)

SEE ALSO

ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating- Point Arithmetic

NAME

fpc, get_fpc_csr, set_fpc_csr, get_tpc_irr, get_fpc_eir, set_fpc_led, swapRM, swapINX – floating-point control registers

SYNOPSIS

#include <sys/fpu.h>

int get_fpc_csr()

int set_fpc_csr(csr)
int csr;

int get_fpc_irr()

int get_fpc_eir()

void set_fpc_led(value)
int value;

int swapRM(x)
int x;

int swapINX(x)
int x;

DESCRIPTION

These routines are to get and set the floating-point control registers of MIPS floating-point units. All of these routines take and or return their values as 32 bit integers.

The file *<sys/fpu.h>* contains unions for each of the control registers. Each union contains a structure that breaks out the bit fields into the logical parts for each control register. This file also contains constants for fields of the control registers.

All implementations of MIPS floating-point have a *control and status* register and a *implementation revision* register. The *control and status* register is returned by *get_fpc_csr*. The routine *set_fpc_csr* sets the *control and status* register and returns the old value. The *implementation revision* register is read-only and is returned by the routine *get_fpc_irr*.

The R2360 floating-point units (floating-point boards) have two additional control registers. The *exception instruction* register is a read-only register and is returned by the routine *get_fpc_eir*. The other floating-point control register on the R2360 is the *leds* register. The low 8 bits corresponds to the leds where a one is off and a zero is on. The *leds* register is a write-only register and is set with the routine *set_fpc_leds*.

The routine *swapRM* sets only the rounding mode and returns the old rounding mode. The routine *swapINX* sets only the sticky inexact bit and returns the old one. The bits in the arguments and return values to *swapRM* and *swapINX* are right justified.

NOTES

swapRM and *swapINX* are in *libm.a* . The link editor searches this library under the ''–lm'' option.

ALSO SEE

R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

NAME

fread, fwrite – binary input/output

SYNOPSIS

#include <stdio.h>

int fread (void *ptr, size_t size, size_t nitems, FILE *stream);

int fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);

DESCRIPTION

fread copies, into an array pointed to by *ptr*, up to *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream*.

fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *fwrite* does not change the contents of the array pointed to by *ptr*.

ferror(3S) may be used to determine if an error occurred in an *fread* or *fwrite* call.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*.

SEE ALSO

read(2), write(2), fopen(3S), ferror(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If successful, both *fread* and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

#include <math.h>

double frexp (double value, int *eptr);

double ldexp (double value, int exp);

double modf (double value, double *iptr);

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \le |x| < 1.0$, and the "exponent" n is an integer. *frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

Ldexp returns the quantity *value* $* \ 2^{exp}$.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±HUGE (defined in **<math.h>**) is returned (according to the sign of *value*), and *errno* is set to **ERANGE**.
If *ldexp* would cause underflow, zero is returned and *errno* is set to **ERANGE**. If *value* is Nan or INF, *frexp ( )* or *modf ( )* raise a floating-point exception.

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>

int fseek (FILE *stream, long int offset, int whence);

void rewind (FILE *stream);

long int ftell (FILE *stream);

DESCRIPTION

fseek sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for **SEEK_SET**, the current value of the file position indicator for **SEEK_CUR**, or end-of-file for **SEEK_END**. A successful call to *fseek* clears the end-of-file indicator for the stream.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, SEEK_CUR), except that the error indicator for the stream is also cleared. Also, *rewind* returns no value.

fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), popen(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on the UNIX system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw (path, fn, depth)
const char *path;
int (*fn) (const char *, struct stat *, int);
int depth;

DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns −1, and sets the error type in *errno*.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

CAVEAT

ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma(double x);

extern int signgam;

DESCRIPTION

gamma returns $\ln|\Gamma(x)|$ where $\Gamma(x) = \int_0^\infty t^{x-1}\, e^{-t}\, dt$ for $x > 0$ and
$\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$ for $x < 1$.

The external integer signgam returns the sign of $\Gamma(x)$.

IDIOSYNCRASIES

Do **not** use the expression signgam*exp(gamma(x)) to compute g := $\Gamma(x)$. Instead use a program like this (in C):

```
lg = gamma(x); g = signgam*exp(lg);
```

Only after *gamma* has returned can *signgam* be correct. Note too that $\Gamma(x)$ must overflow when x is large enough, underflow when –x is large enough, and spawn a division by zero when x is a nonpositive integer.

The following C program fragment might be used to calculate G if the overflow needs to be detected:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
    error();
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to overflow and is defined in the *<values.h>* header file.

Only in the UNIX math library for C was the name *gamma* ever attached to ln G. Elsewhere (in some FORTRAN libraries) the name GAMMA belongs to G and the name ALGAMMA to ln G in single precision; in double the usual names are DGAMMA and DLGAMMA in FORTRAN. Why should C be different?

Some math libraries now define this function as *lgamma*() to suggest its real functionality.

Archaeological records suggest that C's *gamma* originally delivered ln(G (x)). Later, the program *gamma* was changed to cope with negative arguments in a more conventional way, but the documentation did not reflect that change correctly. The most recent change corrects inaccurate values when *x* is almost a negative integer, and lets G (x) be computed without conditional expressions. Programmers should not assume that *gamma* has settled down.

The file *<sgimath.h>*, which is #included in *<math.h>* has the statement:

#define lgamma gamma

so that programs including *<math.h>* automatically transform *lgamma* to *gamma*.

At some point in the future, *gamma* might change its name to *lgamma*.

DIAGNOSTICS

Returns +Infinity for negative integer arguments.

SEE ALSO

math(3M)

NAME

getc, getchar, fgetc, getw – get character or word from a stream

SYNOPSIS

#include <stdio.h>

int getc (FILE *stream);

int getchar (void);

int fgetc (FILE *stream);

int getw (FILE *stream);

DESCRIPTION

getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *getc* and *getchar* are macros.

Fgetc behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Getw returns the next word (i.e., integer) from the named input *stream*. *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

C SYNOPSIS

#include <unistd.h>

char *getcwd (char *buf, int size);

FORTRAN SYNOPSYS

subroutine getcwd (str)
character*128 str

DESCRIPTION

getcwd returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C).
pwd(1) in the *User's Reference Manual.*

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

The possible errors for *getcwd* are:

[EINVAL] The argument *size* is less than or equal to zero.

[ENOMEM] The attempt to malloc a return buffer (because *buf* == NULL) failed.

[ERANGE] The number of bytes in the path string is greater than the *size* specified by the calling routine.

[EACCESS] Read or search permission was denied for a component of the pathname.

NAME

getdtablesize – get descriptor table size

SYNOPSIS

nfds = getdtablesize()
int nfds;

DESCRIPTION

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call *getdtablesize* returns the size of this table.

SEE ALSO

close(2), dup(3C), open(2), select(2)

NAME

getenv – return value for environment name

SYNOPSIS

#include <stdlib.h>

char *getenv (const char *name);

DESCRIPTION

getenv searches the environment list [see *environ*(5)] for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer.

To get the value of the **$HOME** environment variable for example:

```
{
                char *getenv(char *);
                char *name = getenv("HOME");
                /* other processing */
}
```

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent (void);

struct group *getgrgid (gid_t gid);

struct group *getgrnam (const char *name);

void setgrent (void);

void endgrent (void);

struct group *fgetgrent (FILE *f);

DESCRIPTION

getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the *<grp.h>* header file.

```
struct group {
        char    *gr_name;    /* the name of the group */
        char    *gr_passwd;  /* the encrypted group password */
        gid_t   gr_gid;      /* the numerical group ID */
        char    **gr_mem;    /* vector of pointers to member names */
};
```

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete.

fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

NOTE

There are two versions of the primitives documented in this manual entry: a vanilla version and a Yellow Pages version. The programmatic interface of both versions is identical. The vanilla version gets its information from an ASCII file in */etc*. The Yellow Pages version knows about Sun's Yellow Pages distributed lookup service. If you want the Yellow Pages version, link the program according to the instructions for (3Y) primitives as described in *intro*(3). Refer to *ypserv*(1M) and the *NFS User's Guide* for more information about the Yellow Pages.

FILES

/etc/group

SEE ALSO

getgroups(2), getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getgroups – get group access list (Berkeley 4.3 version)

SYNOPSIS

#include <sys/param.h>

ngrps = getgroups(setlen,gidset)
int ngrps, setlen, *gidset;

As described in **intro(3),** in order to link with these BSD-version routines, the compile line must include the following -I and -l specifications:

cc -I/usr/include/bsd prog.c -lbsd

DESCRIPTION

Getgroups retrieves the current group access list of the user process and stores it in the array *gidset*. The parameter *setlen* indicates the number of entries that may be placed in *gidset*. *getgroups* returns the actual number of groups returned in *gidset*. No more than NGROUPS, as defined in *<sys/param.h>*, will ever be returned.

As a special case, if the *setlen* argument is zero, *getgroups()* returns the number of supplemental group IDs associated with the calling process without modifying the array pointed to by the *gidset* argument.

RETURN VALUE

A successful call returns the number of groups in the group set. A value of −1 indicates that an error occurred, and the error code is stored in the global variable *errno* .

ERRORS

The possible errors for *getgroups* are:

[EINVAL] The argument *setlen* is smaller than the number of groups in the group set.

[EFAULT] The argument *gidset* specifies an invalid address.

SEE ALSO

multgrps(1), setgroups(3B), initgroups(3B), getgroups(2), setgroups(2), initgroups(3X)

CAVEATS

This routine adheres to the BSD 4.3 specifications, which differ *markedly* from those of BSD 4.2.

Getgroups(3B) (4.3 BSD version) and getgroups(2) (POSIX version) differ in the types of their *gidset* parameter. These must not be confused.

BUGS

The *gidset* array should be of type **gid_t**, but remains integer for compatibility with earlier BSD systems.

NAME

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent, herror – get network host entry

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const void *addr, int addrlen, int type);

struct hostent *gethostent(void);

void sethostent(int stayopen);

void endhostent(void);

void herror(const char *string);

DESCRIPTION

Gethostbyname and *gethostbyaddr* each return a pointer to a *hostent* data structure describing an Internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, *named*(1M), the Yellow Pages service or broken-out fields from a line in */etc/hosts*.

```
struct   hostent {
         char    *h_name;        /* official name of host */
         char    **h_aliases;    /* alias list */
         int     h_addrtype;     /* host address type */
         int     h_length;       /* length of address */
         char    **h_addr_list;  /* list of addresses from name server */
};
#define  h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name — Official name of the host.

h_aliases — A zero terminated array of alternate names for the host.

h_addrtype — The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in h_addr_list; this is for backward compatibility.

The *name* argument to *gethostbyname* is a character string containing an Internet host name or an Internet address in standard dot notation (see *inet*(3N)). If the name contains no dot, and if the environment variable ''HOSTALIASES'' contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See *hostname*(5) for the alias file format. The *addr* argument to *gethostbyaddr* points to a buffer containing a 32-bit Internet host address in network byte order. *Addrlen* contains the address length in bytes; it should be set to sizeof(struct in_addr). *Type* specifies the address family and should be set to AF_INET.

The *gethostbyname* and *gethostbyaddr* routines can access three types of host-address databases:

- the hosts file, /etc/hosts,
- Yellow Pages (YP) and
- the Berkeley Internet Name Domain service ("BIND name server").

It is possible to specify the query ordering of these databases. As described in *resolver*(4), the system administrator can change the default ordering using the *hostresorder* keyword in */usr/etc/resolv.conf*. Users can override the default with the HOSTRESORDER environment variable.

There are two versions of the routines documented in this manual entry: the standard C library version and the Yellow Pages version in /usr/lib/libsun.a. The programmatic interface of both versions is identical, except for *gethostent*. The standard version is the default; if you want the Yellow Pages version, follow the instructions for (3Y) routines in *intro*(3). The differences are described below.

LIBC VERSION

The default resolution order is: query the BIND name server first, if is not configured, then search */etc/hosts*.

When using the name server, *gethostbyname* will search for the named host in the current domain and its parents unless the name ends in a dot. See *hostname*(5) for the domain search procedure.

Sethostent may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*. Otherwise, queries are

performed using UDP datagrams.

Endhostent closes the TCP connection.

Gethostent is not provided in the standard version.

YP VERSION

If the Yellow Pages is enabled, these routines resolve queries from the YP hosts database. If YP is not enabled, the same query order as the standard version is used.

When Yellow Pages is running, *gethostent* obtains the next entry in the YP *hosts.byaddr* map. *Sethostent* and *endhostent* reset the pointer into the map to the beginning.

If YP is not running, *gethostent* reads the next line of */etc/hosts*, opening the file if necessary. *Sethostent* opens and rewinds the file. If the *stayopen* flag is non-zero, the file will not be closed after each call to *gethostent*. *Endhostent* closes the file.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null (0) pointer. The global integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine *herror* can be used to print an error message to file descriptor 2 (standard error) describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

h_errno can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error.
NO_DATA	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

FILES

/etc/hosts	
/usr/etc/resolv.conf	contains address(es) of remote name server(s)

ENVIRONMENT

HOSTALIASES	contains hostname aliases
HOSTRESORDER	ordering of YP, BIND, and /etc/hosts lookups
LOCALDOMAIN	overrides default domain used by BIND

SEE ALSO

named(1M), resolver(3N), sethostresorder(3N), hosts(4), resolver(4), hostname(5)

Programming and network administration chapters in the *Network Communications Guide*

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

Only the Internet address format is currently understood.

NAME

getinvent, setinvent, endinvent, scaninvent – get hardware inventory entry

SYNOPSIS

#include <invent.h>

inventory_t *getinvent ( )

int setinvent ( )

void endinvent ( )

int scaninvent (fun, arg)
int (*fun)( );
void *arg;

int _keepinvent;

DESCRIPTION

getinvent returns a pointer to an object with the following structure containing an entry from the system hardware inventory table. Each entry in the table contains an "inventory" structure, declared in the *<sys/invent.h>* header file:

```
typedef struct inventory_s {
#ifdef _LANGUAGE_C_PLUS_PLUS
        struct inventory_s *inv_next;
        int     inv_class;
        int     inv_type;
        char    inv_controller;
        char    inv_unit;
        long    inv_state;
#else
        struct inventory_s *next;
        int     class;
        int     type;
        char    controller;
        char    unit;
        long    state;
#endif
} inventory_t;
```

Note: in the next major release, the **inv_**-prefixed, C++ member names will be used by the C language version of this structure.

Each inventory entry is described by a *class* and a class-specific *type*. The remaining fields provide further information on the inventory entry. See the comments in the header file for an explanation of these fields. The *<invent.h>* header file includes *<sys/invent.h>*, and should be included before calling inventory functions.

getinvent when first called returns a pointer to the first inventory structure in the table; thereafter, it returns a pointer to the next inventory structure in the table; so successive calls can be used to search the entire table.

A call to *setinvent* has the effect of rewinding the table to allow repeated searches. *Endinvent* may be called to free allocated storage when processing is complete.

scaninvent applies fun to each inventory entry, passing the entry's address and arg to fun. If fun returns a non-zero value, *scaninvent* stops scanning and returns that value. Otherwise *scaninvent* returns 0 after scanning all entries. *scaninvent* normally calls *endinvent* before returning. To prevent this call, set *_keepinvent* to a non-zero value.

DIAGNOSTICS

getinvent returns a **NULL** pointer when it has read all entries, or if it cannot *setinvent* successfully when first called. *setinvent* returns -1 on failure. *scaninvent* returns -1 if it cannot successfully *setinvent* before scanning.

NAME

getlogin – get login name

SYNOPSIS

#include <unistd.h>

char *getlogin (void);

DESCRIPTION

getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* is not found.

CAVEAT

The return values point to static data whose content is overwritten by each call.

NAME

getlvent, freelvent – get lvtab file entry

SYNOPSIS

#include <sys/lvtab.h>

struct lvtabent *getlvent (FILE *file);

void freelvent (struct lvtabent *tabent);

DESCRIPTION

Getlvent returns a pointer to an object with the following structure declared in the *<lvtab.h>* header file. It is normally used for parsing the */etc/lvtab* file.

```
struct lvtabent     {
        char            *devname;       /* volume device name */
        char            *volname;       /* volume name (human-readable) */
        unsigned        stripe;         /* number of ways striped */
        unsigned        gran;           /* granularity of striping */
        unsigned        ndevs;          /* number of constituent devices */
        int             mindex;         /* not currently used */
        char            *pathnames[1];  /* pathnames of constituent devices */
};
```

The fields have meanings described in *lvtab*(4). Note that the *stripe* field is set to 1 by default if there is no **stripes=** specification in the *lvtab* entry.

Getlvent should be called with a FILE * parameter referencing the file to be parsed; this will usually be */etc/lvtab*. When first called it returns a pointer to an lvtabent structure representing the first entry in the file. Successive calls can be used to search the entire file.

Unlike *getpwent* and *getmntent* the returned pointer points to dynamically allocated memory, since the size of an lvtab entry is very variable. The internal structure of the dynamically allocated space is implementation-dependent, so the function *freelvent* is provided to free the memory when an lvtab entry is no longer needed.

DIAGNOSTICS

If an end-of-file or an error is encountered on reading, *getlvent* returns a NULL pointer. Blank and comment lines in the file are skipped and do not give rise to an lvtabent structure. *Getlvent* attempts to be robust in the face of malformed or nonsensical entries and will coerce the **ndevs** field of the returned lvtabent structure to zero if syntax errors are detected in the entry being parsed. However, the results are not generally defined if the file contents do not conform to the syntax defined in *lvtab*(4).

SEE ALSO

lvtab(4).

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>

struct netent *getnetent(void);

struct netent *getnetbyname(const char *name);

struct netent *getnetbyaddr(long net, int type);

void setnetent(int stayopen);

void endnetent(void);

DESCRIPTION

Getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct    netent {
          char             *n_name;       /* official name of net */
          char             **n_aliases;   /* alias list */
          int              n_addrtype;    /* net number type */
          unsigned long    n_net;         /* net number */
};
```

The members of this structure are:

n_name — The official name of the network.

n_aliases — A zero terminated list of alternate names for the network.

n_addrtype — The type of the network number returned; currently only AF_INET.

n_net — The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of the file, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr*.

Endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

SEE ALSO

networks(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NOTE

There are two versions of the primitives documented in this manual entry: a standard version and a Yellow Pages version. The programmatic interface of both versions is identical. The standard version uses */etc/networks* whereas the YP version uses the Yellow Pages distributed lookup service. The standard version is the default; if you want the Yellow Pages version, follow the instructions for (3Y) routines in *intro*(3). Refer to *ypserv*(1M) and the *NFS User's Guide* for more information about the Yellow Pages.

NAME

getopt – get option letter from argument vector

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see *intro*(1)). So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using *getopt* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

optarg is set to point to the start of the option-argument on return from *getopt*.

getopt places in **optind** the *argv* index of the next argument to be processed. **optind** is external and is initialized to **1** before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **–1**. The special option "--" may be used to delimit the end of the options; when it is encountered, **–1** will be returned, and "--" will be skipped.

DIAGNOSTICS

getopt prints an error message on standard error and returns a question mark (**?**) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting **opterr** to **0**.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the option **o**, which requires an option-argument:

```
main (argc, argv)
int argc;
char **argv;
```

```
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != -1)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                (void)fprintf(stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
        .
        .
}
```

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section above, **a** and **b** are options, and the option **o** requires an option-argument:

cmd –aboxxx file (Rule 5 violation: options with
option-arguments must not be grouped with other options)
cmd –ab –oxxx file (Rule 6 violation: there must be
white space after an option that takes an option-argument)

SEE ALSO

getopts(1), intro(1) in the *User's Reference Manual.*

Changing the value of the variable **optind**, or calling *getopt* with different values of *argv*, may lead to unexpected results.

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

getpass reads up to a newline or EOF from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent(void);

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

void setprotoent(int stayopen);

void endprotoent(void)

DESCRIPTION

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct  protoent {
        char    *p_name;       /* official name of protocol */
        char    **p_aliases;   /* alias list */
        int     p_proto;       /* protocol number */
};
```

The members of this structure are:

p_name   The official name of the protocol.

p_aliases   A zero terminated list of alternate names for the protocol.

p_proto   The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotobyname* or *getprotobynumber*.

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

There are two versions of the primitives documented in this manual entry: a standard version and a Yellow Pages version. The programmatic interface of both versions is identical. The standard version uses */etc/protocols* whereas the YP version uses the Yellow Pages distributed lookup service. The standard version is the default; if you want the Yellow Pages version, follow the instructions for (3Y) routines in *intro*(3). Refer to *ypserv*(1M) and the *NFS User's Guide* for more information about the Yellow Pages.

NAME

getpw – get name from UID

SYNOPSIS

int getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

getpw returns non-zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent (void);

struct passwd *getpwuid (uid_t uid);

struct passwd *getpwnam (const char *name);

void setpwent (void);

void endpwent (void);

struct passwd *fgetpwent (FILE *f);

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file:

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        uid_t   pw_uid;
        gid_t   pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

This structure is declared in *<pwd.h>* so it is not necessary to redeclare it.

The fields have meanings described in *passwd*(4).

getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd.**

NOTE

There are two versions of the primitives documented in this manual entry: a vanilla version and a Yellow Pages version. The programmatic interface of both versions is identical. The vanilla version gets its information from an ASCII file in */etc*. The Yellow Pages version knows about Sun's Yellow Pages distributed lookup service. If you want the Yellow Pages version, link the program according to the instructions for (3Y) primitives as described in *intro*(3). Refer to *ypserv*(1M) and the *NFS User's Guide* for more information about the Yellow Pages.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getrpcent, getrpcbyname, getrpcbynumber – get RPC entry

SYNOPSIS

#include <netdb.h>

struct rpcent *getrpcent(void);

struct rpcent *getrpcbyname(char *name);

struct rpcent *getrpcbynumber(int number);

void setrpcent(int stayopen);

void endrpcent(void);

DESCRIPTION

Getrpcent, *getrpcbyname*, and *getrpcbynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the Sun RPC program number data base, */etc/rpc*, or the Yellow Pages *rpc* map.

```
struct  rpcent {
        char    *r_name;        /* name of server for this rpc program */
        char    **r_aliases;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name — The name of the server for this rpc program.

r_aliases — A zero terminated list of alternate names for the rpc program.

r_number — The rpc program number for this service.

Getrpcent reads the next line of the file, opening the file if necessary.

Setrpcent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getrpcent* (either directly, or indirectly through one of the other ''getrpc'' calls).

Endrpcent closes the file.

Getrpcbyname and *getrpcbynumber* sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc

SEE ALSO

rpc(4), rpcinfo(1M)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getrpcport – get RPC port number

SYNOPSIS

```
int getrpcport(host, prognum, versnum, proto)
        char *host;
        int prognum, versnum, proto;
```

DESCRIPTION

Getrpcport returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

getrusage – get information about resource utilization

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

#define RUSAGE_SELF      0    /* calling process */
#define RUSAGE_CHILDREN -1    /* terminated child processes */

getrusage(who, rusage)
int who;
struct rusage *rusage;
```

DESCRIPTION

Getrusage returns information describing the resources utilized by the current process, or all its terminated child processes. This routine is provided for compatibility with 4.3BSD.

The *who* parameter is one of RUSAGE_SELF or RUSAGE_CHILDREN. The buffer to which *rusage* points will be filled in with the following structure:

```
struct rusage {
        struct timeval ru_utime;   /* user time used */
        struct timeval ru_stime;   /* system time used */
        int     ru_maxrss;
        int     ru_ixrss;          /* integral shared text memory size */
        int     ru_idrss;          /* integral unshared data size */
        int     ru_isrss;          /* integral unshared stack size */
        int     ru_minflt;         /* page reclaims */
        int     ru_majflt;         /* page faults */
        int     ru_nswap;          /* swaps */
        int     ru_inblock;        /* block input operations */
        int     ru_oublock;        /* block output operations */
        int     ru_msgsnd;         /* messages sent */
        int     ru_msgrcv;         /* messages received */
        int     ru_nsignals;       /* signals received */
        int     ru_nvcsw;          /* voluntary context switches */
        int     ru_nivcsw;         /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime   the total amount of time spent executing in user mode.

ru_stime	the total amount of time spent in the system executing on behalf of the process(es).

The remaining fields are not maintained by the IRIX kernel and are set to zero by this routine.

ERRORS

The possible errors for *getrusage* are:

[EINVAL]	The *who* parameter is not a valid value.
[EFAULT]	The address specified by the *rusage* parameter is not in a valid part of the process address space.

SEE ALSO

gettimeofday(3), wait(2)

BUGS

There is no way to obtain information about a child process that has not yet terminated.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets (char *s);

char *fgets (char *s, int n, FILE *stream);

DESCRIPTION

gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

fgets reads characters from the *stream* into the array pointed to by *s*, until $n-1$ characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

void setservent(int stayopen);

void endservent(void);

DESCRIPTION

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name — The official name of the service.

s_aliases — A zero terminated list of alternate names for the service.

s_port — The port number at which the service resides. Port numbers are returned as a 16-bit value in network byte order.

s_proto — The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getservbyname* or *getservbyport*.

Endservent closes the file.

Getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3N), services(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32-bit quantity is probably naive.

NOTE

There are two versions of the primitives documented in this manual entry: a standard version and a Yellow Pages version. The programmatic interface of both versions is identical. The standard version uses */etc/services* whereas the YP version uses the Yellow Pages distributed lookup service. The standard version is the default; if you want the Yellow Pages version, follow the instructions for (3Y) routines in *intro*(3). Refer to *ypserv*(1M) and the *NFS User's Guide* for more information about the Yellow Pages.

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

#include <sys/time.h>

int gettimeofday(struct timeval *tp, struct timezone *tzp);

int settimeofday(struct timeval *tp, struct timezone *tzp);

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday* call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. By default, the resolution for *gettimeofday* is 100 HZ (which equals 10 milliseconds). Use *ftimer*(1) to increase the resolution of *gettimeofday*.

In System V, the time zone used by each process is determined by the TIMEZONE environment variable. The *tzp* argument is present here only for compatibility. It cannot be used to set the time zone for the system, and so must be zero for settimeofday. If *tzp* is not zero, gettimeofday will return an interpretation of the TIMEZONE environment variable.

The structures pointed to by *tp* and *tzp tzp* are defined in *<sys/time.h>* as:

```
struct timeval {
        long    tv_sec;          /* seconds since Jan. 1, 1970 */
        long    tv_usec;         /* and microseconds */
};

struct timezone {
        int     tz_minuteswest;  /* of Greenwich */
        int     tz_dsttime;      /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day or time zone.

These routines emulate the 4.3BSD system calls.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

[EPERM] A user other than the super-user attempted to set the time.

[EINVAL] An attempt to change the timezone was made.

SEE ALSO

date(1), time(2), stime(2), ctime(3C)

See ftimer(1), lboot(1M), mpadmin(1) for details on changing the resolution.

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

#include <utmp.h>

struct utmp *getutent (void);

struct utmp *getutid (struct utmp *id);

struct utmp *getutline (struct utmp *line);

struct utmp *pututline (struct utmp *utmp);

void setutent (void);

void endutent (void);

void utmpname (const char *file);

DESCRIPTION

getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char      ut_user[8];        /* User login name */
        char      ut_id[4];          /* /etc/inittab id (usually line #) */
        char      ut_line[12];       /* device name (console, lnxx) */
        short     ut_pid;            /* process id */
        short     ut_type;           /* type of entry */
        struct    exit_status {
          short     e_termination;   /* Process termination status */
          short     e_exit;          /* Process exit status */
        } ut_exit;                   /* The exit status of a process
                                      * marked as DEAD_PROCESS. */
        time_t    ut_time;           /* time entry was made */
};
```

getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the

pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

getwd – get current working directory pathname

SYNOPSIS

char *getwd(pathname)
char *pathname;

DESCRIPTION

Getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

LIMITATIONS

Maximum pathname length is PATH_MAX characters as defined in <*limits.h*>. PATH_MAX is equivalent to MAXPATHLEN as defined in <*sys/param.h*>.

DIAGNOSTICS

Getwd returns zero and places a message in *pathname* if an error occurs.

NAME

handle_sigfpes – floating-point exception handler package

SYNOPSIS

```
#include <sigfpe.h>

void handle_sigfpes (int onoff, int en_mask,
                     void (*user_routine)(unsigned long),
                     int abort_action,
                     void (*abort_routine)(unsigned[5], int[2]));

struct sigfpe_template
        {
        int repls;
        int count;
        int trace;
        int abort;
        int exit;
        };
extern struct sigfpe_template sigfpe_[_N_EXCEPTION_TYPES+1];

int invalidop_results_[_N_INVALIDOP_RESULTS+1];

int invalidop_operands_[_N_INVALIDOP_OPERANDS+1];
```

DESCRIPTION

The MIPS floating-point accelerator may raise floating-point exceptions due to five conditions: **_OVERFL**(*overflow*), **_UNDERFL**(*underflow*), **_DIVZERO**(*divide-by-zero*), **_INEXACT**(*inexact result*), or **_INVALID**(*invalid operand*, e.g., infinity). Usually these conditions are masked, and do not cause a floating-point exception. Instead, a default value is substituted for the result of the operation, and the program continues silently. This event may be intercepted by causing an exception to be raised. Once an exception is raised, the specific conditions which caused the exception may be determined, and more appropriate action taken.

The library **libfpe.a** provides two methods to unmask and handle these conditions: the subroutine **handle_sigfpes**, and the environment variable **TRAP_FPE**. Both methods provide a mechanism for unmasking each of these conditions except **_INEXACT**, for *handling* and classifying exceptions arising from them, and for substituting either a default value or a chosen one. They also provide mechanisms to count, trace, exit or abort on enabled exceptions. The subroutine **handle_sigfpes** will always override options set by the environment variable **TRAP_FPE**. **TRAP_FPE** is supported for Fortran, C and Pascal. **Handle_sigfpes** is supported for C and Fortran.

Arguments to **handle_sigfpes** have the following interpretation:

onoff is a flag indicating whether handling is being turned on *(onoff == _ON)* or off *(onoff == _OFF)*. Information from the sigfpe structure will be printed if *(onoff == _DEBUG)*. (defined in *sigfpe.h*).

en_mask indicates which of the four conditions should be unmasked, enabling them to raise floating-point exceptions. *en_mask* is only valid if onoff == _ON, and is the bitwise or of the constants _EN_UNDERFL, _EN_OVERFL, _EN_DIVZERO, and _EN_INVALID (defined in *sigfpe.h*).

user_routine: **handle_sigfpes** provides a mechanism for setting the result of the operation to any one of a set of well-known values. If full control over the value of selected operations is desired for one or more exception conditions, a function *user_routine* must be provided. For these selected exception conditions, *user_routine* will be called to set the value resulting from the operation.

abort_action: If the handler encounters an unexpected condition, an inconsistency, or begins looping, the flag *abort_action* indicates what action should be taken. Legal values are:

_TURN_OFF_HANDLER_ON_ERROR	instruct the floating-point-accelerator to cease causing exceptions and continue. (i.e., disable handling)
_ABORT_ON_ERROR	kill the process after giving an error message and possibly calling a user-supplied cleanup routine.
_REPLACE_HANDLER_ON_ERROR	install the indicated user routine as the handler when such an error is encountered. Future floating-point exceptions will branch to the user-routine. (see signal(2))

abort_routine: When a fatal error (i.e., one described under *abort_action* above) is encountered, *abort_routine* is used as the address of a user routine. If abort_action is _ABORT_ON_ERROR, and *abort_routine* is valid, it is called before aborting, and passed a pointer to the address of the instruction causing the exception as its single argument. (*see below under* DIAGNOSTICS) If *abort_action* is _REPLACE_HANDLER_ON_ERROR, and *abort_routine* is valid, it will be installed as the new handler. In this case, the instruction which caused

the unexpected exception will be re-executed, causing a new exception, and *abort_routine* entered. (see signal(2))

When an exception is encountered, the handler examines the instruction causing the exception, the state of the floating-point accelerator and the sigfpe structure to determine the correct action to take, and the program is continued. In the cases of **_UNDERFL, _OVERFL, _DIVZERO,** and some instances of **_INVALID,** an appropriate value is substituted for the result of the operation, and the instruction which caused the exception is skipped. For most exceptions arising due to an invalid operand (**_INVALID** exceptions), more meaningful behavior may be obtained by replacing an erroneous operand. For these conditions, the operand is replaced, and the instruction re-issued.

sigfpe: For each enabled exception, the sigfpe structure contains the fields: repls, count, trace, exit and abort. For each enabled exception <p>, and each non-zero entry <n> in the sigfpe structure, the trap handler will take the following actions:
count: A count of all enabled traps will be printed to stderr at the end of execution of the program , and every at <n>th exception <p>.
trace: A dbx stack trace will be printed to stderr every execption <p>, up to <n> times.
abort: Core dump and abort program upon encountering the <n>th exception <p>. The abort option takes precedence over the exit option.
exit: Exit program upon encountering the <n>th exception <p>.
repls: Each of the exceptions **_UNDERFL**, **_OVERFL**, and **_DIVZERO** has an associated default value which is used as the result of the operation causing the exception. These default values may be overridden by initializing this integer value. This value is interpreted as an integer code used to select one of a set of replacement values, or to indicate that the routine *user_routine* is responsible for setting the value. These integer codes are listed below:

_ZERO	use zero as the replacement value
_MIN	use the appropriately-typed minimum value as the replacement. (i.e., the smallest number which is representable in that format *without* denormalizing)
_MAX	use the appropriately-typed maximum value as the replacement
_INF	use the appropriately-typed value for infinity as the replacement
_NAN	use the appropriately-typed value for not-a-number as the replacement. (A *quiet* not-a-number is used.)
_APPROPRIATE	use a handler-supplied appropriate value as the replacement. These are different from the default values: _ZERO for **UNDERFL**, _MAX for **_OVERFL**, _INF for **_DIVZERO**. Values for **_INVALID** are handled on a case-by-case basis.
_USER_DETERMINED	invoke the routine *user_routine* (see note) to set the value of the operation. If this is the code used for **_INVALID** exceptions, all such exceptions will defer to *user_routine* to set their value. In this case, *invalidop_results_* and *invalidop_operands_* will be ignored.

The default values used as the results of floating-point exceptions are:

values for *sigfpe_.repls*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	
1	_UNDERFL	underflow	_MIN
2	_OVERFL	overflow	_MAX
3	_DIVZERO	divide-by-zero	_MAX
4	_INVALID	invalid operand	_APPROPRIATE

For **_INVALID** exceptions, the correct action may be either to set the result and skip the instruction, or to replace an operand and retry the instruction. There are four cases in which the result is set. The array named *invalidop_results_* is consulted for replacement codes for these cases:

array *invalidop_results_*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	
1	_MAGNITUDE_INF_SUBTRACTION	$\infty - \infty$	_INF
2	_ZERO_TIMES_INF	$0 * \infty$	_ZERO
3	_ZERO_DIV_ZERO	0/0	_ZERO
4	_INF_DIV_INF	∞ / ∞	_INF

There are six cases in which an offending operand is replaced. An array named *invalidop_operands_* is consulted for user-initialized codes for these cases. Each element governs the following cases:

array *invalidop_operands_*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	(none)
1	_SQRT_NEG_X	sqrt(-x)	(currently not supported)
2	_CVT_OVERFL	conversion to float caused target to overflow	_MAX
3	_TRUNK_OVERFL	conversion to integer caused target to overflow	_MAX
4	_CVT_NAN	conversion of NaN	_MAX
5	_CVT_INF	conversion of ∞	_MAX
6	_UNORDERED_CMP	comparison to NaN	_MAX
7	_SNAN_OP	operand was Signaling Nan	_MAX

NOTE

Use of *user_routine* to set values

If the integer code defining the replacement value for a particular exception condition is _USER_DEFINED, the user-supplied routine *user_routine* is called:

(*user_routine)(exception_parameters, value);

value is an array of two *int*s into which *user_routine* should store the replacement value.
exception_parameters is an array of five *unsigned int*s which describe the exception condition:

array *exception_parameters*		
#	element mnemonic	description
0	_EXCEPTION_TYPE	the exception type (_DIVZERO, etc).
1	_INVALID_ACTION	value = _SET_RESULT if result is being set. Otherwise, an operand is being replaced. This element is meaningful only if the exception type is **_INVALID**.
2	_INVALID_TYPE	This element is meaningful only if the exception type is **_INVALID** It is the index corresponding to the particular conditions giving rise to the exception. In conjunction with element 1, this value uniquely determines the exception condition. (e.g., if _INVALID_ACTION is _SET_RESULT and _INVALID_TYPE is 2, the **_INVALID** exception is due to _ZERO_TIMES_INF.)
3	_VALUE_TYPE	the type of the replacement value - either _SINGLE, _DOUBLE or _WORD.
4	_VALUE_SIGN	the suggested sign *user_routine* should use for the replacement value - either _POSITIVE or _NEGATIVE.

The environment variable **TRAP_FPE**:

If the code has been compiled with **libfpe.a**, the runtime startup routine will check for the environment variable "**TRAP_FPE**". The string read as the value of **TRAP_FPE** will be interpreted and **handle_sigfpes** will be called with the resulting values. If the program contains an explicit call to handle_sigfpes, that call will override all actions defined by **TRAP_FPE**.

TRAP_FPE is read in upper case letters only. The string assigned to **TRAP_FPE** may be in upper case or lower case. **TRAP_FPE** can take one of two forms: either a global value, or a list of individual items.

global values:

"" or OFF	Execute the program with no trap handling enabled. Same as **TRAP_FPE** undefined. Same as linking without libfpe.a
ON	Same as **TRAP_FPE="ALL=DEFAULT"**.

Alternately, replacement values and actions may be specified for each of the possible trap types individually. This is accomplished by setting the environment variable as follows:

setenv **TRAP_FPE** "item;item;item...."

an item can be one of the following:

traptype=statuslist	Where traptype defines the specific floating point exception to enable, and statuslist defines the list of actions upon encountering the trap.
DEBUG	Confirm the parsing of the environment variable, and the trap actions.

Traptype can be one of the following literal strings:

UNDERFL	underflow
OVERFL	overflow
DIVZERO	divide by zero
INVALID	invalid operand
ALL	all of the above

Statuslist is a list seperated by commas. It contains an optional symbolic replacement value, and an optional list of actions.

symbolic replacement values:

DEFAULT	Do not override the predefined default values.
IEEE	Maps to integer code _APPROPRIATE.
ZERO	Maps to integer code _ZERO.
MIN	Maps to integer code _MIN.
MAX	Maps to integer code _MAX.
INF	Maps to integer code _INF.
NAN	Maps to integer code _NAN.

All actions take an optional integer in parentheses:

Note: for any traps that have an action and no specified replacement value, the DEFAULT replacement value will be used.

COUNT(n)	A count of the trap type will be printed to stderr every nth trap, and at the end of the program. Default is MAXINT.
ABORT(n)	Core dump and abort the program upon encountering the nth trap. Default id 1.
EXIT(n)	Exit program upon encountering the nth trap. Default id 1.
TRACE(n)	If a trap is encountered, Print a stack trace to stderr up to n times. Default is 10.

EXAMPLE

setenv **TRAP_FPE** "ALL=COUNT; UNDERFL=ZERO; OVERFL=IEEE,TRACE(5), ABORT(100); DIVZERO=ABORT"

Count all traps, trace the first five overflows, abort on the first divide by zero, or the 100th overflow. Replace zero for underflows, the "appropriate" value for overflows, and the default values for divide by zero, and invalid operands.

SEE ALSO

signal(3c), fsigfpe(3f)

DIAGNOSTICS

If the handler encounters an unexpected condition, an inconsistency, or begins looping, the flag *abort_action* and function address *abort_routine* (parameters to **handle_sigfpes**) indicate what action should be taken. If abort_action is _ABORT_ON_ERROR, the handler will be removed leaving the exceptions enabled, an error message printed, and the instruction

causing the fault re-issued, giving a core dump. Prior to this, if *abort_routine* is valid, it is invoked as

(*abort_routine)(ptr_to_pc);

where *ptr_to_pc* is a pointer to the address of the instruction which caused the exception.

If *abort_action* is _REPLACE_HANDLER_ON_ERROR, and *abort_routine* is valid, **handle_sigfpes** removes its handler and installs *abort_routine* as the new handler. The instruction which caused the exception will be re-executed, causing a new exception, and *abort_routine* entered. (see signal(2))

If *abort_action* is _TURN_OFF_HANDLER_ON_ERROR **handle_sigfpes** will mask (disable) floating-point exceptions and remove its handler. The instruction which caused the fault will then be re-issued, continuing the program as if floating-point exceptions had never been enabled.

Any other combination of the two parameters *abort_action* and *abort_routine* will cause **handle_sigfpes** to remove its handler, generate an error message, and re-issue the instruction causing the exception, producing a core dump.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (ENTRY item, ACTION action);

int hcreate (unsigned nel);

void hdestroy (void);

DESCRIPTION

hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the *<search.h>* header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to *hcreate*.

BUGS

Hsearch is compiled by Silicon Graphics with none of the flags named in NOTES defined.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* [see *string*(3C)].

CHAINED

Use a linked list to resolve collisions. If this option is selected, the following other options become available.

START Place new entries at the beginning of the linked list (default is at the end).

SORTUP Keep the linked list sorted by key in ascending order.

SORTDOWN

Keep the linked list sorted by key in descending order.

The source code should be consulted for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>

struct info {              /* this is the info stored in the table */
        int age, room;     /* other than the key. */
};
#define NUM_EMPL    5000   /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

        /* create table */
        (void) hcreate(NUM_EMPL);
        while (scanf("%s%d%d", str_ptr, &info_ptr->age,
```

```
                &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                  /* put info in structure, and structure in item */
                  item.key = str_ptr;
                  item.data = (char *)info_ptr;
                  str_ptr += strlen(str_ptr) + 1;
                  info_ptr++;
                  /* put item into table */
                  (void) hsearch(item, ENTER);
          }

          /* access table */
          item.key = name_to_find;
          while (scanf("%s", item.key) != EOF) {
              if ((found_item = hsearch(item, FIND)) != NULL) {
                  /* if item is in the table */
                  (void)printf("found %s, age = %d, room = %d\n",
                          found_item->key,
                          ((struct info *)found_item->data)->age,
                          ((struct info *)found_item->data)->room);
              } else {
                  (void)printf("no such employee %s\n",
                          name_to_find)
              }
          }
  }
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

hsearch and *hcreate* use *malloc*(3C) to allocate space.

CAVEAT

Only one hash search table may be active at any given time.

NAME

hypot, cabs – Euclidean distance, complex absolute value

SYNOPSIS

#include <math.h>

double hypot (double x, double y);

float fhypot (float x, float y);

double cabs (struct { double a,b; } z);

float fcabs (struct { float a,b; } z);

DESCRIPTION

mypot(x,y), *fhypot(y,y)*, *cabs(x,y)*, and *fcabs(x,y)* return sqrt(x*x+y*y) computed in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

fhypot and *fcabs* are the same functions as *hypot* and *cabs* but for the float data type.

hypot(∞,v) = *hypot*(v,∞) = +∞ for all v, including *NaN*.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns +infinity

ERROR (due to Roundoff, etc.)

Below 0.97 *ulps*. Consequently *hypot*(5.0,12.0) = 13.0 exactly; in general, *hypot* and *cabs* return an integer whenever an integer might be expected.

NOTES

As might be expected, *hypot*(v,*NaN*) and *hypot*(*NaN*,v) are *NaN* for all *finite* v. Programmers might be surprised at first to discover that hypot(±∞,*NaN*) = +∞. This is intentional; it happens because hypot(∞,v) = +∞ for *all* v, finite or infinite. Hence hypot(∞,v) is independent of v. The IEEE *NaN* is designed to disappear when it turns out to be irrelevant, as it does in hypot(∞,*NaN*).

SEE ALSO

math(3M), sqrt(3M)

AUTHOR

W. Kahan

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *cp);

unsigned long inet_network(char *cp);

char *inet_ntoa(struct in_addr in);

struct in_addr inet_makeaddr(int net, int lna);

unsigned long inet_lnaof(struct in_addr in);

unsigned long inet_netof(struct in_addr in);
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard ''.'' (dot) notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in ''.'' notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the ''.'' notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as ''128.net.host''.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as ''net.host''.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as ''parts'' in a ''.'' notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant **INADDR_NONE** is returned by *inet_addr* and *inet_network* for malformed requests.

SEE ALSO

gethostbyname(3N), getnetent(3N), hosts(4), networks(4)

BUGS

The problem of host byte ordering versus network byte ordering is confusing.

A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed.

The string returned by *inet_ntoa* resides in a static memory area.

Inet_addr should return a struct in_addr.

NAME

initgroups – initialize group access list

SYNOPSIS

POSIX:

#include <sys/types.h>

int initgroups(char *name, gid_t basegid);

BSD:

int initgroups(char *name, int basegid);

To use the BSD versions of *setgroups*, *getgroups*, or *initgroups*, you must either

1) explicitly invoke them as *BSDsetgroups*, *BSDgetgroups*, or *BSDinitgroups*, or

2) link with the libbsd.a library:

```
cc -o prog prog.c -lbsd
```

DESCRIPTION

initgroups reads through the group file and sets up, using the appropriate version of the *setgroups* call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is the group number from the password file.

The 4.3BSD version and the POSIX version differ in the type of the *basegid* parameter.

FILES

/etc/group
/etc/passwd

SEE ALSO

multgrps(1), getgroups(2), setgroups(2), sysconf(2)

DIAGNOSTICS

initgroups returns 0 if successful, otherwise –1 and the error code is stored in global integer *errno*.

ERRORS

The *initgroups* call will fail if:

[EPERM] The caller is not the super-user.

The maximum number of groups to which any individual user may belong is determined differently for the POSIX and BSD versions of *initgroups*. BSD limits the user to **NGROUPS** groups, as defined in *<sys/param.h>*. POSIX allows this maximum to be alterable at system boot-time and therefore provides the *sysconf(*_SC_NGROUPS_MAX*)* system call--which *initgroups*

utilizes--to determine the value at runtime. (The value is actually set in /usr/sysgen/master.d/kernel.) If the invoking user is a member of too many groups, *initgroups* will display an appropriate message on *stderr* and initialize as many as allowed from **/etc/group**.

BUGS

initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

NAME

initgroups – initialize group access list (BSD 4.3 version)

SYNOPSIS

#include <sys/types.h>

initgroups(name, basegid)
char *name;
gid_t basegid;

As described in **intro(3)**, in order to link with these BSD-version routines, the compile line must include the following -I and -l specifications:

cc -I/usr/include/bsd prog.c -lbsd

DESCRIPTION

Initgroups reads through the group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

FILES

/etc/group

SEE ALSO

multgrps(1), getgroups(3B), setgroups(3B), getgroups(2), setgroups(2), initgroups(3X)

DIAGNOSTICS

Initgroups returns 0 if successful, otherwise −1 and the error code is stored in extern int *errno*.

ERRORS

The *initgroups* call will fail if:

[EPERM] The caller is not the super-user.

If the invoking user is a member of too many groups (more than NGROUPS, as defined in <sys/param.h>), *initgroups* will display an appropriate message on *stderr* and initialize only the first NGROUPS groups, as listed in /etc/group.

BUGS

Initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

Insque and *remque* manipulate queues built from doubly-linked lists. Each element in the queue must in the form of ''struct qelem''. *Insque* inserts *elem* in a queue immediately after *pred*; *remque* removes an entry *elem* from a queue.

SEE ALSO

''VAX Architecture Handbook'', pp. 228-235.

NAME

kill – send signal to a process (4.3BSD)

SYNOPSIS

#include <signal.h>

int kill(int pid, int sig);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<signal.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

kill sends the signal *sig* to a process, specified by the process number *pid*. *Sig* may be one of the signals specified in *sigvec*(3B), or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user. A single exception is the signal **SIGCONT**, which may always be sent to any descendant of the current process.

If the process number is 0, the signal is sent to all processes in the sender's process group; this is a variant of *killpg*(3B).

If the process number is –1 and the user is the super-user, the signal is broadcast universally except to system processes. If the process number is –1 and the user is not the super-user, the signal is broadcast universally to all processes with the same uid as the user. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not –1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of *killpg*(3B).

Processes may send signals to themselves.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

kill will fail and no signal will be sent if any of the following occur:

[EINVAL]	*Sig* is not a valid signal number.
[ESRCH]	No process can be found corresponding to that specified by *pid.*
[ESRCH]	The process id was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and its effective user ID does not match the effective user ID of the receiving process.

SEE ALSO

getpid(2), getpgrp(2), killpg(3B), sigvec(3B)

CAVEATS (IRIX)

When the process number is −1, the process sending the signal is NOT included in the delivery group. In the IRIX implementation, the sending process receives the signal, too.

4.3BSD's implementation of kill returns **EPERM** if any members of a process group can not be signaled (when *kill* is invoked with a *pid* of 0). The IRIX implementation does not.

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

killpg – send signal to a process group (4.3BSD)

SYNOPSIS

#include <signal.h>

int killpg(int pgrp, int sig);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<signal.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

killpg sends the signal *sig* to the process group *pgrp*. See *sigvec*(3B) for a list of signals.

The sending process and members of the process group must have the same effective user ID, or the sender must be the super-user. As a single special case the continue signal **SIGCONT** may be sent to any process that is a descendant of the current process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

killpg will fail and no signal will be sent if any of the following occur:

[EINVAL]	*Sig* is not a valid signal number.
[ESRCH]	No process can be found in the process group specified by *pgrp*.
[ESRCH]	The process group was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

kill(3B), getpgrp(2), sigvec(3B)

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

l3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;

DESCRIPTION

If **TYPE**(*ldptr*) is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead.*

Ldahread returns **SUCCESS** or **FAILURE.** If **TYPE**(*ldptr*) does not represent an archive file or if it cannot read the archive header, *Ldahread* fails.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ar(4), ldfcn(4), and intro(4).

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;

DESCRIPTION

ldopen(3X) and *ldclose* provide uniform access to simple object files and object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If **TYPE**(*ldptr*) does not represent an archive file, *ldclose* closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number for an archive file and if archive has more files, *ldclose* reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a later *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

Ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE**(*ldptr*). *Ldaclose* always returns **SUCCESS**. The function is often used with *ldaopen*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

BUGS

ONLY the memory associated with the **LDFILE** structure proper is freed. Memory allocated for subordinate structures (e.g. the symbol table) must be freed **PRIOR** to calling *ldclose* or *ldaclose*.

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* . It reads the file header into the area of memory beginning at *filehead.*

Ldfhread returns **SUCCESS** or **FAILURE.** If *ldfhread* cannot read the file header, it fails.

Usually, *ldfhread* can be avoided by using the macro **HEADER(***ldptr***)** defined in **<ldfcn.h>** (see *ldfcn*(4)). Note that the information in HEADER is swapped, if necessary. The information in any field, *fieldname*, of the file header can be accessed using **HEADER(***ldptr***)**.*fieldname*.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetaux – retrieve an auxiliary entry, given an index

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

pAUXU ldgetaux (ldptr, iaux)
LDFILE ldptr;
long iaux;

DESCRIPTION

Ldgetaux returns a pointer to an auxiliary table entry associated with *iaux*. The AUXU is contained in a static buffer. Because the buffer can be overwritten by later calls to *ldgetaux*, it must be copied by the caller if the aux is to be saved or changed.

Note that auxiliary entries are not swapped as this routine cannot detect what manifestation of the AUXU union is retrieved. If LDAUXSWAP(ldptr, ldf) is non-zero, a further call to *swap_aux* is required. Before calling the *swap_aux* routine, the caller should copy the aux.

If the auxiliary cannot be retrieved, *Ldgetaux* returns **NULL** (defined in **<stdio.h>**) for an object file. This occurs when:

- the auxiliary table cannot be found
- the iaux offset into the auxiliary table is beyond the end of the table

Typically, *ldgetaux* is called immediately after a successful call to *ldtbread* to retrieve the data type information associated with the symbol table entry filled by *ldtbread.* The index field of the symbol, pSYMR, is the *iaux* when data type information is required. If the data type information for a symbol is not present, the index field is *indexNil* and ldgetaux should not be called.

The program must be loaded with the object file access routine library *libmld.a.*

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldgetname – retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE ldptr;
pSYMR symbol;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer. Because the buffer can be overwritten by later calls to *ldgetname*, the caller must copy the buffer if the name is to be saved.

If the name cannot be retrieved, *ldgetname* returns **NULL** (defined in **<stdio.h>**) for an object file. This occurs when:

- the string table cannot be found
- the name's offset into the string table is beyond the end of the string table

Typically, *ldgetname* is called immediately after a successful call to *ldtbread. Ldgetname* retrieves the name associated with the symbol table entry filled by *ldtbread.*

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldgetpd – retrieve procedure descriptor given a procedure descriptor index

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

long ldgetpd (ldptr, ipd, ppd)
LDFILE ldptr;
long ipd;
pPDR ipd;

DESCRIPTION

Ldgetpd returns a SUCCESS or FAILURE depending on whether the procedure descriptor with index *ipd* can be accessed. If it can be accessed, the structure pointed to by *ppd* is filled with the contents of the corresponding procedure descriptor. The *isym, iline,* and *iopt* fields of the procedure descriptor are updated to be used in further LD routine calls. The *adr* field is updated from the symbol referenced by the *isym field.*

The PDR cannot be retrieved when:

- The procedure descriptor table cannot be found.
- The ipd offset into the procedure descriptor table is beyond the end of the table.
- The file descriptor that the ipd offset falls into cannot be found.

Typically, *ldgetpd* is called while traversing the table that runs from 0 to SYMHEADER(ldptr).ipdMax - 1.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINER *linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINER *linent;

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, which is the index of its local symbols entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together do exactly the same function as *ldlread.* After an initial call to *ldlread* or *ldlinit, ldlitem* can be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply finds the line number entries for the function identified by *fcnindx. Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE.** If no line number entries exist in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum, ldlread* fails. If no line number entries exist in the object file or if *fcnindx* does not index a function entry in the symbol table, *ldlinit* fails. If it finds no line number equal to or greater than *linenum,*

ldlitem fails.

The programs must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnlseek seeks to the line number entries of the section specified by *sectname*.

Ldlseek and *ldnlseek* return SUCCESS or FAILURE. **NOTE:** Line numbers are not associated with sections in the MIPS symbol table; therefore, the second argument is ignored, but maintained for historical purposes.

If they cannot seek to the specified line number entries, both routines fail.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

Ldohseek returns **SUCCESS** or **FAILURE**. If the object file has no optional header or if it cannot seek to the optional header, *ldohseek* fails.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;

LDREADST (ldptr, flags)
LDFILE *ldptr;
int flags;

DESCRIPTION

Ldopen and *ldclose*(3X) provide uniform access to simple object files and to object files that are members of archive files. An archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, *ldopen* opens *filename*, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and **TYPE**(*ldptr*) is the archive magic number, *ldopen* reinitializes the **LDFILE** structure for the next archive member of *filename*.

Ldopen and *ldclose* work in concert. *Ldclose* returns **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, and particularly when a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose:*

```
/* for each filename to be processed */

ldptr = NULL;
do {
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, *ldaopen* opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the fields from *oldptr*. *Ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers can be used concurrently to read separate parts of the object file. For example, one pointer can be used to step sequentially through the relocation information while the other is used to read indexed symbol table entries.

Ldopen and *ldaopen* open *filename* for reading. If *filename* cannot be opened or if memory for the **LDFILE** structure cannot be allocated, both functions return **NULL**. A successful open does not ensure that the given file is a common object file or an archived object file.

Ldopen causes the symbol table header and file descriptor table to be read. Further access, using *ldptr*, causes other appropriate sections of the symbol table to be read (for example, if you call *ldtbread, the symbols or externals are read). To force sections of the symbol table into memory, call ldreadst* with *ST_P** constants ORed together from *<cmplrs/stsupport.h>.*

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname:

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return SUCCESS or FAILURE. If *sectindx* is greater than the number of sections in the object file, *ldrseek* fails; if there is no section name corresponding with *sectname, ldnrseek* fails. If the specified section has no relocation entries or if it cannot seek to the specified relocation entries, either function fails.

NOTE: The first section has an index of *one*.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return **SUCCESS** or **FAILURE.** If *sectindx* is greater than the number of sections in the object file, *ldshread* fails; If there is no section name corresponding with *sectname, ldnshread* fails. If it cannot read the specified section header, either function fails.

NOTE: The first section header has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnsseek seeks to the section specified by *sectname*.

Ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. If *sectindx* is greater than the number of sections in the object file, *ldsseek* fails; if there is no section name corresponding with *sectname*, *ldnsseek* fails. If there is no section data for the specified section or if it cannot seek to the specified section, either function fails.

NOTE: The first section has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;

DESCRIPTION

Ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* can be used in later calls to *ldtbread*(3X). *Ldtbindex* returns the index of the last *ldtbread*.

If there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry, *Ldtbindex* fails and returns BADINDEX (-1).

NOTE: The first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
pSYMR symbol;

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns **SUCCESS** or **FAILURE**. If *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry, *ldtbread* fails.

The local and external symbols are concatenated into a linear list. Symbols are accessible from symnum zero to *SYMHEADER(ldptr).isymMax+SYMHEADER(ldptr).iextMax.* The index and iss fields of the SYMR are made absolute (rather than file relative) so that routines *ldgetname*(3X), *ldgetaux*(3X), and ldtbread (this routine) proceed normally given those indices. Only the ''sym'' part of externals is returned.

NOTE: The first symbol in the symbol table has an index of *zero*.

The program must allocate space big enough for a **SYMR** and point the **pSYMR** argument at the space before calling *ldtbread.*

The program must be loaded with the object file access routine library **libmld.a.**

SEE ALSO

ldclose(3X), ldgetname(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4).

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns **SUCCESS** or **FAILURE**. If the symbol table has been stripped from the object file or if it cannot seek to the symbol table, *ldtbseek* fails.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

lockf – record locking on files

SYNOPSIS

#include <unistd.h>

int lockf (int fildes, int function, long size);

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

Fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

Function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define    F_ULOCK    0    /* Unlock a previously locked section */
#define    F_LOCK     1    /* Lock a section for exclusive use */
#define    F_TLOCK    2    /* Test and lock a section for exclusive use */
#define    F_TEST     3    /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

Size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to

the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a −1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]
: *Fildes* is not a valid open descriptor.

[EACCES]
: *Cmd* is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
: *Cmd* is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

logname returns a pointer to the null-terminated login name; it extracts the **LOGNAME** environment variable from the user's environment.

This routine is kept in **/usr/lib/libPW.a.**

FILES

/etc/profile

SEE ALSO

getenv(3C), profile(4), environ(5).
env(1), login(1) in the *User's Reference Manual.*

CAVEATS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

#include <stdio.h>
#include <search.h>

```
void *lsearch ((const void *)key, (void *)base,
               size_t nmemb, size_t size,
               int (*compar)(const void *, const void *));

void *lfind ((const void *)key, (void *)base,
             size_t nmemb, size_t size,
             int (*compar)(const void *, const void *));
```

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. **Key** points to the datum to be sought in the table. **Base** points to the first element in the table. **Nmemb** points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *Size* is the size of the key in bytes (*sizeof (*key)*). **Compar** is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

Lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120
```

```
char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
unsigned nel = 0;
int strcmp( );
. . .
while (fgets(line, ELSIZE, stdin) != NULL &&
   nel < TABSIZE)
        (void) lsearch(line, (char *)tab, &nel,
                  ELSIZE, strcmp);
. . .
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

m_fork, m_kill_procs, m_set_procs, m_get_numprocs, m_get_myid, m_next, m_lock, m_unlock, m_park_procs, m_rele_procs, m_sync – parallel programming primitives

C SYNOPSIS

#include <ulocks.h>
#include <task.h>

int m_fork (void (*func)(), ...);

int m_kill_procs (void);

int m_set_procs (int numprocs);

int m_get_numprocs (void);

int m_get_myid (void);

unsigned m_next (void);

void m_lock (void);

void m_unlock (void);

int m_park_procs (void);

int m_rele_procs (void);

void m_sync (void);

FORTRAN SYNOPSIS

integer*4 function m_fork (func, [arg1, arg2, arg3, arg4, arg5, arg6])
external func
integer*4 arg1, arg2, arg3, arg4, arg5, arg6

integer*4 function m_kill_procs ()

integer*4 m_set_procs (numprocs)
integer*4 numprocs

integer*4 function m_get_numprocs ()

integer*4 function m_get_myid ()

integer*4 function m_next ()

subroutine m_lock ()

subroutine m_unlock ()

integer*4 function m_park_procs ()

integer*4 function m_rele_procs ()

subroutine m_sync ()

DESCRIPTION

The *m_fork* routine creates *n-1* processes that execute the given *func* in parallel with the calling process. The processes are created using the *sproc(2)* system call, and share all attributes (virtual address space, file descriptors, uid, etc.). Once the processes are all created, they each start executing the subprogram *func* taking up to six arguments as passed to *m_fork*. The arguments passed must not be larger than pointers in size, i.e. floating point numbers must be passed by reference. The processes execute the subprogram and wait until they all return at which point the *m_fork* call returns.

The number of subtasks *n* can be set and queried using *m_set_procs* and *m_get_numprocs*, where the default is the number of processors in the system (and hence the maximum number of processes that can truly be run in parallel).

When the processes are created, each is assigned a unique identifier called the tid, for thread id. This identifier can be obtained through *m_get_myid*. Thread id's range from 0 to n-1.

A global counter and a global lock are provided to simplify synchronization between the processes. On each *m_fork* call, the counter is reset to zero. The counter value is gotten and post incremented through the *m_next* routine. The first time *m_next* is called, it returns a zero. The global lock is set and unset through *m_lock* and *m_unlock*.

The *m_park_procs* and *m_rele_procs* are provided to suspend and resume the child processes created by *m_fork*. This is useful if you have a phase of the program where the parent will do setup or reinitialization code and you do not want to have the children spinning and wasting resources. *m_park_procs* should not be called when processes are already suspended.

m_sync is provided to synchronize all threads at some point in the code. When *m_sync* is called by each thread, it waits at that point for all other threads to call *m_sync*. The global counter is reset, and all threads resume after the m_sync call.

m_kill_procs terminates the processes created from the previous *m_fork*.

NOTES

These primitives are based on the Sequent Computer Systems parallel programming primitives, but may not conform to all Sequent semantics.

RETURN VALUE

m_fork, *m_set_procs*, *m_park_procs*, *m_rele_procs*, and *m_kill_procs* return a 0 when successful, and a -1 with errno set upon failure. *m_get_numprocs*, *m_get_myid*, and *m_next* all return integers. *m_lock*,

m_unlock, and *m_sync* return no value.

SEE ALSO

sproc(2), blockproc(2), prctl(2), barrier(3P), usinit(3P), ussetlock(3P).

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc (size_t size);

void free (void *ptr);

void *realloc (void *ptr, size_t size);

void *calloc (size_t nelem, size_t elsize);

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* [see *brk*(2)] to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

NAME

math – introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library *libm*. There are three versions of the math library *libm.a*, *libm43.a* and *libfastm.a*

The first, *libm.a*, contains routines written in MIPS assembly language and tuned for best performance and includes many routines for the *float* data type. The routines in there are based on the algorithms of Cody and Waite or those in the 4.3BSD release, whichever provides the best performance with acceptable error bounds. Those routines with Cody and Waite implementations are marked with a '*' in the list of functions below.

The second version of the math library, *libm43.a*, contains routines all based on the original codes in the 4.3BSD release. The difference between the two version's error bounds is typically around 1 unit in the last place, whereas the performance difference may be a factor of two or more.

The link editor searches this library under the "–lm" (or "–lm43") option. Declarations for these functions may be obtained from the include file <*math.h*>. The FORTRAN math library is described in "man 3f intro".

The third library, *libfastm.a*, contains only faster, lower-precision versions of *sqrt*(3m) and *fsqrt*(3m).

LIST OF FUNCTIONS

The cycle counts of all functions are approximate; cycle counts often depend on the value of argument. The error bound sometimes applies only to the primary range.

List of Functions

Name	Appears on Page	Description	Error Bound (ULPs)		Cycles	
			libm.a	libm43.a	libm.a	libm43.a
acos	sin.3m	inverse trigonometric function	3	3	?	?
acosh	asinh.3m	inverse hyperbolic function	3	3	?	?
asin	sin.3m	inverse trigonometric function	3	3	?	?
asinh	asinh.3m	inverse hyperbolic function	3	3	?	?
atan	sin.3m	inverse trigonometric function	1	1	152	260
atanh	asinh.3m	inverse hyperbolic function	3	3	?	?
atan2	sin.3m	inverse trigonometric function	2	2	?	?
cabs	hypot.3m	complex absolute value	1	1	?	?
cbrt	sqrt.3m	cube root	1	1	?	?

ceil	floor.3m	integer no less than	0	0	?	?
copysign	ieee.3m	copy sign bit	0	0	?	?
cos*	sin.3m	trigonometric function	2	1	128	243
cosh*	sinh.3m	hyperbolic function	?	3	142	294
drem	ieee.3m	remainder	0	0	?	?
erf	erf.3m	error function	?	?	?	?
erfc	erf.3m	complementary error function	?	?	?	?
exp*	exp.3m	exponential	2	1	101	230
expm1	exp.3m	exp(x)–1	1	1	281	281
fabs	floor.3m	absolute value	0	0	?	?
fatan*	sin.3m	inverse trigonometric function	3		64	
fcos*	sin.3m	trigonometric function	1		87	
fcosh*	sinh.3m	hyperbolic function	?		105	
fexp*	exp.3m	exponential	1		79	
flog*	exp.3m	natural logarithm	1		100	
floor	floor.3m	integer no greater than	0	0	?	?
fsin*	sin.3m	trigonometric function	1		68	
fsinh*	sinh.3m	hyperbolic function	?		44	
fsqrt	sqrt.3m	square root	1		95	
ftan*	sin.3m	trigonometric function	?		61	
ftanh*	sinh.3m	hyperbolic function	?		116	
hypot	hypot.3m	Euclidean distance	1	1	?	?
j0	j0.3m	bessel function	?	?	?	?
j1	j0.3m	bessel function	?	?	?	?
jn	j0.3m	bessel function	?	?	?	?
gamma	gamma.3m	log gamma function	?	?	?	?
log*	exp.3m	natural logarithm	2	1	119	217
logb	ieee.3m	exponent extraction	0	0	?	?
log10*	exp.3m	logarithm to base 10	3	3	?	?
log1p	exp.3m	log(1+x)	1	1	269	269
pow	exp.3m	exponential x**y	60–500	60–500	?	?
rint	floor.3m	round to nearest integer	0	0	?	?
scalb	ieee.3m	exponent adjustment	0	0	?	?
sin*	sin.3m	trigonometric function	2	1	101	222
sinh*	sinh.3m	hyperbolic function	?	3	79	292
sqrt	sqrt.3m	square root	1	1	133	133
tan*	sin.3m	trigonometric function	?	3	92	287
tanh*	sinh.3m	hyperbolic function	?	3	156	293
y0	j0.3m	bessel function	?	?	?	?
y1	j0.3m	bessel function	?	?	?	?
yn	j0.3m	bessel function	?	?	?	?

NOTES

In 4.3BSD, distributed from the University of California in late 1985, most of the foregoing functions come in two versions, one for the double-precision "D" format in the DEC VAX−11 family of computers, another for double-precision arithmetic conforming to the IEEE Standard 754 for Binary Floating-point Arithmetic. The two versions behave very similarly, as should be expected from programs more accurate and robust than was the norm when UNIX was born. For instance, the programs are accurate to within the numbers of *ulps* tabulated above; an *ulp* is one *U*nit in the *L*ast *P*lace. And the programs have been cured of anomalies that afflicted the older math library *libm* in which incidents like the following had been reported:

sqrt(−1.0) = 0.0 and log(−1.0) = −1.7e38.
cos(1.0e−11) > cos(0.0) > 1.0.
pow(x,1.0) ≠ x when x = 2.0, 3.0, 4.0, ..., 9.0.
pow(−1.0,1.0e10) trapped on Integer Overflow.
sqrt(1.0e30) and sqrt(1.0e−30) were very slow.

MIPS machines conform to the IEEE Standard 754 for Binary Floating-point Arithmetic, to which only the notes for IEEE floating-point apply and are included here.

IEEE STANDARD 754 Floating-point Arithmetic:

This standard is on its way to becoming more widely adopted than any other design for computer arithmetic.

The main virtue of 4.3BSD's *libm* codes is that they are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine that conforms to IEEE 754 need use anything worse than the new *libm*.

Properties of IEEE 754 Double-precision:

Wordsize: 64 bits, 8 bytes. Radix: Binary.
Precision: 53 significant bits, roughly like 16 significant decimals.
If x and x' are consecutive positive Double-precision numbers (they differ by 1 *ulp*), then
1.1e−16 < 0.5**53 < (x'−x)/x ≤ 0.5**52 < 2.3e−16.
Range: Overflow threshold = 2.0**1024 = 1.8e308
Underflow threshold = 0.5**1022 = 2.2e−308
Overflow goes by default to a signed ∞.
Underflow is *Gradual,* rounding to the nearest integer multiple of 0.5**1074 = 4.9e−324.
Zero is represented ambiguously as +0 or −0.
Its sign transforms correctly through multiplication or

division, and is preserved by addition of zeros with like signs; but x−x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,±0). In particular, comparison (x > y, x ≥ y, etc.) cannot be affected by the sign of zero; but if finite x = y then ∞ = 1/(x−y) ≠ −1/(y−x) = −∞.

∞ is signed.

it persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/±∞ = ±0 (nonzero)/0 = ±∞. But ∞−∞, ∞*0 and ∞/∞ are, like 0/0 and sqrt(−3), invalid operations that produce *NaN*. ...

Reserved operands:

there are 2**53−2 of them, all called *NaN* (*N*ot *a* *N*umber). Some, called Signaling *NaN*s, trap any floating-point operation performed upon them; they could be used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaN*s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x ≠ x then x is *NaN*; every other predicate (x > y, x = y, x < y, ...) is FALSE if *NaN* is involved.

NOTE: Trichotomy is violated by *NaN*.

Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, −, *, /, √) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards +∞ or towards −∞ at the programmer's option. And the same kinds of rounding are specified for Binary−Decimal Conversions, at least for magnitudes between roughly 1.0e−10 and 1.0e37.

Exceptions:

IEEE 754 recognizes five kinds of floating-point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	*NaN*, or FALSE
Overflow	$\pm\infty$
Divide by Zero	$\pm\infty$
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE: An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating-point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1) Test for a condition that might cause an exception later, and branch to avoid the exception.

2) Test a flag to see whether an exception has occurred since the program last reset its flag.

3) Test a result to see whether it is a value that only an exception could have produced.
CAUTION: The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be

missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

4) ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error–handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

— No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

— In a subprogram that lacks an error-handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error-handling statement is encountered or the whole task is aborted and memory is dumped.

5) STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

6) ... Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3BSD's *libm*. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

i) No exception should be signaled that is not deserved by the data supplied to that function.

ii) Any exception signaled should be identified with that function rather than with one of its subroutines.

iii) The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves/restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
: when a result, if properly computed, might have lain barely within range, and

Inexact in *cabs*, *cbrt*, *hypot*, *log10* and *pow*
: when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
: any result but *NaN* would probably be misleading.

Overflow is signaled only when
: the exact result would be finite but beyond the overflow threshold.

Divide–by–Zero is signaled only when
: a function takes exactly infinite values at finite operands.

Underflow is signaled only when
: the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
: greater range or precision would be needed to represent the exact result.

Exceptions on MIPS machines:

The exception enables and the flags that are raised when an exception occurs (as well as the rounding mode) are in the floating–point control and status register. This register can be read or written by the routines described on the man page *fpc*(3C). This register's layout is described in the file *<sys/fpu.h>*.

A full implementation of IEEE 754 "user trap handlers" is under development at MIPS computer systems. At which time all functions in *libm* will appear atomic and the full functionality of user trap handlers will be supported in thoses language without other

floating–point error handling intrinsics (i.e. Ada, PL/1, etc). For a description of these trap handlers see section 8 of the IEEE 754 standard.

What is currently available is only the raw interface which was only intended to be used by the code to implement IEEE user trap handlers. IEEE floating-point exceptions are enabled by setting the enable bit for that exception in the floating-point control and status register. If an exception then occurs the UNIX signal SIGFPE is sent to the process. It is up to the signal handler to determine the instruction that caused the exception and to take the action specified by the user. The instruction that caused the exception is in one of two places. If the floating-point board is used (the floating-point implementation revision register indicates this in its implementation field) then the instruction that caused the exception is in the floating-point exception instruction register. In all other implementations the instruction that caused the exception is at the address of the program counter as modified by the branch delay bit in the cause register. Both the program counter and cause register are in the sigcontext structure passed to the signal handler (see *signal*(2)). If the program is to be continued past the instruction that caused the exception the program counter in the signal context must be advanced. If the instruction is in a branch delay slot then the branch must be emulated to determine if the branch is taken and then the resulting program counter can be calculated (see *emulate_branch*(3X) and *signal*(2)).

BUGS

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine–trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

SEE ALSO

signal(2), fpc(3C), emulate_branch(3X)
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description
An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix– and Word–length–independent Standard for Floating–point Arithmetic" by W. J. Cody et al. Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded

drafts of the standard.

AUTHOR

W. Kahan, with the help of Z–S. Alex Liu, Stuart I. McDonald, Dr. Kwok–Choi Ng, Peter Tang.

NAME

memory: memchr, memcmp, memcpy, memset, memccpy – memory operations

SYNOPSIS

#include <string.h>

void *memchr (const void *s, int c, size_t n);

int memcmp (const void *s1, const void *s2, size_t n);

void *memcpy (void *s1, const void *s2, size_t n);

void *memset (void *s, int c, size_t n);

void *memccpy (void *s1, const void *s2, int c, int n);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

Memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.

Memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

Memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

Memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

All of these functions are declared in the *<string.h>* header file.

CAVEATS

Memcmp is implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp, mkstemp – make a unique file name

SYNOPSIS

#include <stdio.h>

char *mktemp (char *template);

int mkstemp(char *template);

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* will replace the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

Mkstemp makes the same replacement to the template but returns a file descriptor for the template file open for reading and writing. *Mkstemp* avoids the race between testing whether the file exists and opening it for use.

The *mkstemp* routine is from the 4.3BSD standard C library.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

DIAGNOSTICS

Mktemp will assign to *template* the NULL string if it cannot create a unique name.

Mkstemp returns an open file descriptor upon success. It returns –1 if no suitable file could be created.

CAVEAT

If called more than 17,576 times in a single process, this function will start recycling previously used names.

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

#include <mon.h>
#include <cmplrs/prof_header.h>

int monitor(int (*lowpc)(), int (*highpc)(),
_WORD *buffer, unsigned bufsize, unsigned nfunc);

void monstartup(int (*lowpc)(), int (*highpc)());

void moncontrol(int mode);

DESCRIPTION

NOTE: These functions have been moved from the standard C library to the *libprof1* library. If a program needs to access these routines it must either use the *-p* option on the compiler/linker or explicitly link with the **-lprof1** library.

Use of the option *-p* during compilation and linking (see *The MIPS Languages Programmer Guide*) automatically generates calls to the *monitor*, *monstartup*, and *moncontrol* functions. You need to call these functions explicitly only if you want finer control over profiling.

There are three varieties of profiling available: program-counter (pc) sampling, invocation counting, and basic block counting. The functions described on this page provide only pc-sampling, the *pixie*(1) command must be used to get the other types of profiling information.

The *-p* option used during linking forces the link editor (*ld*) to include a special start-up routine *mcrt1.o* and the library *libprof1.a* that contains these routines.

monstartup allocates space using *malloc*(3) and passes the space to *monitor* (see below) to provide room for profile data in memory during execution. *monstartup* also specifies the range of addresses for program counter sampling, starting with *lowpc* and ending just below *highpc*. The space allocated provides for pc-sampling at a 2 instruction grain, that is, each pair of instructions (8 bytes) is counted in a single bucket. *monstartup* is normally called automatically at program startup time by *mcrt1.o*.

Without *mcrt1.o* , to profile the entire program, use:

```
extern int (*etext)(), (*eprol)();
. . .
monstartup(eprol, etext);
```

etext lies just above all program text, as described in *end*(3c).

To stop execution monitoring and write the results in an output file, use:

```
monitor(0);
```

This is done automatically by a special *exit* function linked in with *mcrt1.o*.

moncontrol selectively disables and re-enables pc-sampling within a program. To disable pc-sampling, use:

```
moncontrol (_SUSPEND_PROF);
```

to resume, use:

```
moncontrol (_RESUME_PROF);
```

This allows the cost of particular operations to be measured. If the **PROFDIR** variable is not set as described below, *moncontrol* cannot enable pc-sampling; if any profiling is enabled, *moncontrol* cannot prevent the program from generating a file of profiling information on exit.

monitor is a low-level interface to *profil*(2). *lowpc* is the address of the lowest function to be pc-sampled; *highpc* is the address of the lowest function not to be pc-sampled; and *buffer* is the address of a (user supplied) array of *bufsize* short integers. The *nfunc* parameter is no longer used. The size of *buffer* determines the grain of the pc-sampling.

To profile the entire program with a grain of **GRAIN** instructions, use:

```
extern int (*eprol)(), (*etext)();
...
bufsize = sizeof(struct prof_header) +
          (((etext - eprol) / GRAIN) / sizeof(int *))
          * sizeof(_WORD);
monitor(eprol, etext, buf, bufsize, 0);
```

The buffer also contains a profiling header, so space should be reserved for it also: (sizeof(struct prof_header)).

The location of the profiling output files, and whether or not calls to *monitor* will cause pc-sampling to be started are determined by the environment variable **PROFDIR**. If **PROFDIR** is not set, the results will be placed in a file called *mon.out* in the current directory (unless, as explained below, shared address processes are being pc-sampled). If **PROFDIR** is set to a nonempty string, it constructs a file name of the form "profdir/pid.progname", where "profdir" comes from the environment variable, "pid" is the process id, and "progname" is the "argv[0]" for the process.

It is also possible via *moncontrol* to profile parts of a program, write those results to a file, and continue profiling. The

moncontrol (_NEW_PROF_PHASE);

function causes the current contents of the profiling buffer to be written to a file of the form "profdir/pid.progname.phase_id" or "mon.out.phase_id"; where "phase_id" starts at 1 and increments for each call to *moncontrol*. The first file name does not contain any "phase_id". The profiling buffer is then cleared.

If a program that is performing pc-sampling executes the *fork*(2) system call, the profiling information is duplicated, and each will continue to pc-sample into their own buffer. However, it is important that **PROFDIR** be defined otherwise the last process to exit will overwrite the values in *mon.out* of the rest.

If a program that is performing pc-sampling executes the *sproc*(2) system call, *monstartup* is called from *mcrt1.o* to initiate profiling for the new process. At exit time, regardless of whether **PROFDIR** is set, unique file names will be created.

FILES

mon.out	default name for output file
libprof1.a	routines for pc-sampling
/usr/lib/mcrt1.o	special start-up routine for pc-sampling

SEE ALSO

cc(1), pixie(1), prof(1), ld(1), fork(2), profil(2), sproc(2), malloc(3), end(3c) and *The MIPS Languages Programmer Guide.*

DIAGNOSTICS

monitor returns 0 on failure due to insufficient memory or *bufsize* being too small. It returns 1 for a successful call. If the result file cannot be created or written to, an error message is printed on stderr and a 0 is returned. *monstartup* forces the caller to exit on a failed call to *monitor*.

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier *dbm*(3B) library, which managed only a single database.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbm_open*. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see *open*(2)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE. DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *Dbm_firstkey* will return the first key in the database. *Dbm_nextkey* will return the next key in the database. This code will traverse the data base:

> **for** (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))

Dbm_error returns non-zero when an error has occurred reading or writing the database. *Dbm_clearerr* resets the error condition on the named database.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*. If *dbm_store* called with a *flags* value of **DBM_INSERT** finds an existing entry with the same key it returns 1.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Dbm_store* will return an error in the event that a disk block fills with inseparable data.

Dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

SEE ALSO

dbm(3B)

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

nlist(const char *filename, struct nlist *nl);

cc ... -lmld

DESCRIPTION

NOTE: The *nlist* subroutine has moved from the standard C library to the "mld" library due to the difference in the object file format. Programs that need to use *nlist* must be linked with the **–lmld** option.

nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, n_type is set to 1 and the value of the name is inserted in the n_value field. If the name is not found, both entries are set to 0.

The entire symbol table is searched sequentially starting with the external symbols. In case there are multiple occurrences of the same name the value of the first such name found is taken.

See *a.out*(4) for an introduction to the executable file layout.

This subroutine is useful for examining the system name list kept in the file **/unix.**

SEE ALSO

a.out(4)

DIAGNOSTICS

If the file cannot be found or if it is not a valid namelist –1 is returned; otherwise, the number of unfound namelist entries is returned.

If the file is stripped, -1 is returned.

CAVEAT

The order to search the symbol table and the types of names visible to *nlist* are not specified in standard documentation.

NAME

oserror, setoserror – get/set system error

C SYNOPSIS

#include <errno.h>

int oserror(void);

int setoserror(const int err);

FORTRAN SYNOPSIS

integer*4 function oserror()

integer*4 function setoserror(err)
integer*4 err;

DESCRIPTION

oserror returns the last error encountered when performing a system call (or certain library calls). Possible errors are listed in *errno.h.* The value returned is the same as the one in the global *errno* except that if shared processes are performing system calls simultaneously, *errno* may be overwritten, but *oserror* will always return the correct value for the specific process.

setoserror may be used to set the process specific error value. This is primarily used by library routines.

RETURN VALUE

Both *oserror* and *setoserror* return the current system error.

SEE ALSO

intro(2), sproc(2), perror(3C)

NAME

pcreate: pcreatel, pcreatev, pcreateve, pcreatelp, pcreatevp – create a process

SYNOPSIS

int pcreatel (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int pcreatev (path, argv)
char *path, *argv[ ];

int pcreateve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int pcreatelp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int pcreatevp (file, argv)
char *file, *argv[ ];

DESCRIPTION

pcreate in all its forms creates a new process and then transforms that process into the requested process. These routines are totally equivalent to a *fork()* and *exec()* pair EXCEPT that calling process does not incur any extra virtual space penalty. During a normal *fork()/exec()* sequence, the calling process actually duplicates, thus requiring, temporarily, twice as much virtual space. The *exec()* then removes that space and starts the new process. However, a very large program may not be allowed to *fork()* due to insufficient backing store (swap area). *pcreate* may be used to circumvent this problem. The calling process is duplicated via *sproc(2)* then the new process is formed via *exec(2)*.

SEE ALSO

fork(2), exec(2), prctl(2), sproc(2).

DIAGNOSTICS

all diagnostics are from either *sproc(2)* or *exec(2)*.

NAME

perror, strerror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
#include <stdio.h>

void perror (const char *s);

#include <string.h>

char *strerror (int errnum);

#include <errno.h>

extern int errno;
extern char *sys_errlist [ ];
extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. (However, if s is NULL or an empty string, the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, *strerror* takes an error number and returns the corresponding message string. The array of message strings *sys_errlist* is also provided; *errno* can be used as an index into this table to get the message string without the new-line. *Sys_nerr* is the number of messages in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2), oserror(3C).

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (const char *command, const char *type);

int pclose (FILE *stream);

DESCRIPTION

popen creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. *Command* consists of a shell command line. *Type* is an I/O mode, either **r** for reading or **w** for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

EXAMPLE

A typical call may be:

```
char *cmd = "ls *.c";
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
     while (fgets(buf, n, ptr) != NULL)
          (void) printf("%s ",buf);
```

This will print in *stdout* [see *stdio* (3S)] all the file names in the current directory that have a ".c" suffix.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), stdio(3S), system(3S).

DIAGNOSTICS

popen returns a NULL pointer if files or processes cannot be created.

Pclose returns –1 if *stream* is not associated with a "*popen*ed" command.

WARNING

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush* [see *fclose*(3S)].

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

#include <stdio.h>

int printf (const char *format, ...);

int fprintf (FILE *stream, const char *format, ...);

int sprintf (char *s, const char *format, ...);

DESCRIPTION

printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output," followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its arguments (represented in the synopsis by ...) under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by

the field width.

An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer argument. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (*) instead of a digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the arguments specifying field width or precision must appear *before* the argument (if any) to be converted. A negative field width argument is taken as a '–' flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x or X** conversion, a non-zero result will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,u,x,X The integer argument is converted to signed decimal (**d** or **i**), unsigned octal, (**o**), decimal (**u**), or hexadecimal notation (**x** or **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double argument is converted to decimal notation in the style "[–]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double argument is converted in the style "[–]d.ddde±dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double argument is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character argument is printed.

s The argument is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for the argument will yield undefined results.

% Print a %; no argument is converted.

In printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is

[-]NaN0xdddddddd

where 0xdddddddd is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is

[±]inf.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

psignal, sys_siglist – system signal messages

SYNOPSIS

psignal(sig, s)
unsigned sig;
char *s;

extern char *sys_siglist[];

DESCRIPTION

Psignal produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in <*signal.h*>.

To simplify variant formatting of signal names, the vector of message strings *sys_siglist* is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in <*signal.h*> is the number of messages provided for in the table.

SEE ALSO

signal(2), perror(3C)

NAME

psio – NeWS buffered input/output package

SYNOPSIS

#include "psio.h"

PSFILE *psio_stdin;
PSFILE *psio_stdout;
PSFILE *psio_stderr;

DESCRIPTION

The functions described here constitute a user-level I/O buffering scheme for use when communicating with NeWS. This package is based on the standard I/O package that comes with Unix. The functions in this package are used in the same way as the similarly named functions in Standard I/O.

The in-line macros *psio_getc* and *psio_putc* handle characters quickly. The higher level routines *psio_read*, *psio_printf*, *psio_fprintf*, *psio_write* all use or act as if they use *psio_getc* and *psio_putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **PSFILE.** *psio_open* creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **psio.h** include file and associated with the standard open files:

psio_stdin standard input file
psio_stdout
standard output file
psio_stderr
standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams.

Any module that uses this package must include the header file of pertinent macro definitions, as follows:

#include "psio.h"

The functions and constants mentioned in here are declared in that header file and need no further declaration. The constants and the following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc*, *putc*, *psio_eof*, *psio_error*, *psio_fileno*, and *psio_clearerr*.

SEE ALSO

open(2V), close(2), read(2V), write(2V), intro(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), getc(3S), printf(3S), putc(3S), ungetc(3S).
4Sight Programmer's Guide, section 2 "Programming in NeWS," chapter N-8 "Mixing PostScript and C"

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **PSFILE** pointer has not been initialized with *psio_open*, input (output) has been attempted on an output (input) stream, or a **PSFILE** pointer designates corrupt or otherwise unintelligible **PSFILE** data.

LIST OF FUNCTIONS

Name	*Description*
void psio_clearerr(PSFILE*)	stream status inquiries
int psio_close(PSFILE*)	flush a stream
int psio_eof(PSFILE*)	stream status inquiries
int psio_error(PSFILE*)	stream status inquiries
PSFILE *psio_fdopen(int,char*)	open a stream
int psio_flush(PSFILE*)	close or flush a stream
int psio_fileno(PSFILE*)	stream status inquiries
void psio_fprintf(PSFILE*,char*,...)	formatted output conversion
int psio_getc(PSFILE*)	get character or integer from stream
PSFILE* psio_open(char*,char*)	open a stream
int psio_read(char*,int,int,PSTREAM*)	buffered binary input/output
void psio_printf(char*,...)	formatted output conversion
void psio_putc(char,PSFILE*)	put character or word on a stream
void psio_ungetc(char, PSFILE*)	push character back into input stream
int psio_write(char*,int,int,PSFILE*)	buffered binary input/output

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

#include <stdio.h>

int putc (int c, FILE *stream);

int putchar (int c);

int fputc (int c, FILE *stream);

int putw (int w, FILE *stream);

DESCRIPTION

putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *putchar*(*c*) is defined as *putc*(*c, stdout*). *putc* and *putchar* are macros.

fputc behaves like *putc*, but is a function rather than a macro. *fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

putw writes the word (i.e. integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *putw* neither assumes nor causes special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

On success, these functions (with the exception of *putw*) each return the value they have written. [*putw* returns *ferror (stream)*]. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot grow. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

CAVEATS

Because it is implemented as a macro, *putc* evaluates a *stream* argument more than once. In particular, **putc(c, *f++);** doesn't work sensibly. *fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

String points to a string of the form *"name=value." putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

WARNINGS

putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3C) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.

A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

#include <pwd.h>

int putpwent (const struct passwd *p, FILE *f);

DESCRIPTION

putpwent is the inverse of *getpwent*(3C). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd.**

SEE ALSO

getpwent(3C).

DIAGNOSTICS

putpwent returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts (const char *s);

int fputs (const char *s, FILE *stream);

DESCRIPTION

puts writes the null-terminated string pointed to by *s* ,followed by a new-line character, to the standard output stream *stdout.*

fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

NOTES

puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

#include <stdlib.h>

void qsort ((void *base, size_t nel, size_t size,
int (*compar)(const void *, const void *));

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Size* is the size of an element in bytes (*sizeof (*base)*). *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C).
sort(1) in the *User's Reference Manual.*

NAME

raise – send signal to executing program

SYNOPSIS

#include <signal.h>

int raise (int sig);

DESCRIPTION

raise sends the signal *sig* to the calling process using *kill*(2).

SEE ALSO

kill(2), signal(2), sigaction(2).

DIAGNOSTICS

raise returns zero if the signal was successfully sent, –1 otherwise. See *kill*(2) for possible errors.

NAME

rand, srand – simple random-number generator

SYNOPSIS

#include <stdlib.h>

int rand (void);

void srand (unsigned int seed);

DESCRIPTION

rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of *rand* are limited. *drand48*(3C) and *random*(3C) provide a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C), random(3C)

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

#include <math.h>

long random(void);

int srandom(int seed);

char *initstate(unsigned int seed, char *state, int n);

char *setstate(char *state);

DESCRIPTION

Random uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16\times(2^{31}-1)$.

Random/srandom have (almost) the same calling sequence and initialization properties as *rand/srand.* The difference is that *rand*(3C) produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, ''random()&01'' will produce a random binary value.

Srandom does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like *rand*(3C), however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with *1* as the seed.

The *initstate* routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. *Initstate* returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. *Setstate* returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate.*

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed). The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

drand48(3C), rand(3C)

BUGS

About 2/3 the speed of *rand*(3C).

AUTHOR

Earl T. Cohen

NAME

ranhashinit, ranhash, ranlookup – access routine for the symbol table definition file in archives

SYNOPSIS

#include <ar.h>

int ranhashinit(pran, pstr, size)
struct ranlib *pran;
char *pstr;
int size;

ranhash(name)
char *name;

struct ranlib *ranlookp(name)
char *name;

DESCRIPTION

Ranhashinit initializes static information for future use by *ranhash* and *ranlookup. Pran* points to an array of ranlib structures. *Pstr* points to the corresponding ranlib string table (these are only used by *ranlookup). Size* is the size of the hash table and should be a power of 2. If the size isn't a power of 2, a 1 is returned; otherwise, a 0 is returned.

Ranhash returns a hash number given a name. It uses a multiplicative hashing algorithm and the *size* argument to *ranhashinit.*

Ranlookup looks up *name* in the ranlib table specified by *ranhashinit.* It uses the *ranhash* routine as a starting point. Then, it does a rehash from there. This routine returns a pointer to a valid ranlib entry on a match. If no matches are found (the "emptiness" can be inferred if the ran_off field is zero), the empty ranlib structure hash table should be sparse. This routine does not expect to run out of places to look in the table. For example, if you collide on all entries in the table, an error is printed to stderr and a zero is returned.

The program must be loaded with the object file access routine library **libmld.a.**

AUTHOR

Mark I. Himelstein

SEE ALSO

ar(1).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
int inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;

DESCRIPTION

Rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *Rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *Ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(1M) server (among others).

Rcmd looks up the host **ahost* using *gethostbyname*(3N), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd*(1M).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged Internet ports are those in the range 512 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostbyaddr*(3N) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the local user's home directory to see if the request for service is allowed. A 0 is returned if the machine name is listed in the ''hosts.equiv'' file, or the host and remote user name are found in the ''.rhosts'' file; otherwise *ruserok* returns −1. If the *superuser* flag is 1, the checking of the ''hosts.equiv'' file is bypassed. If the local domain (as obtained from *gethostname* (2)) is the same as the remote domain, only the machine name need be specified.

SEE ALSO

rlogin(1C), rsh(1C), intro(2), rexec(3N), rexecd(1M), rlogind(1M), rshd(1M)

DIAGNOSTICS

Rcmd returns a valid socket descriptor on success. It returns -1 on error and prints a diagnostic message on the standard error.

Rresvport returns a valid, bound socket descriptor on success. It returns -1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean ''All network ports in use.''

NAME

readv – read input to scattered buffers

SYNOPSIS

#include <sys/types.h>
#include <sys/uio.h>

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;

DESCRIPTION

Readv attempts to read from the object referenced by the descriptor *d*, scattering the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt−1]. The *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *Readv* will always fill an area completely before proceeding to the next.

On objects capable of seeking, *readv* starts at a position given by the pointer associated with *d* (see *lseek*(2)). Upon return from *readv*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, *readv* return the number of bytes actually read and placed in the *iovec* buffers. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

If the returned value is 0, then end-of-file has been reached.

RETURN VALUE

If successful, the number of bytes actually read is returned. Otherwise, a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Readv will fail if one or more of the following are true:

[EBADF] *D* is not a valid file or socket descriptor open for reading.

[EFAULT] Part of the *iov* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from the file system.

[EINTR] A read from a slow device was interrupted before any data arrived by the delivery of a signal. *d* was negative.

[EAGAIN] The file was a *stream* marked for non-blocking I/O, and no data were ready to be read.

[EWOULDBLOCK]
The file was a socket marked for non-blocking I/O, and no data were ready to be read.

CAVEAT

Readv is implemented using *read*(2), and may ignore errors. If some data are read in the course of a *readv* call, but a *read* error occurs, the call returns the number of bytes successfully read, hiding the error. It is assumed that a subsequent call will discover persistent errors, and that sporadic errors such as EWOULDBLOCK can be ignored.

SEE ALSO

dup(2), fcntl(2), open(2), pipe(2), read(2), select(2), socket(2)

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

DESCRIPTION

regcmp compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. *malloc*(3C) is used to create space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their meaning in *ed*(1).

$ Matches the end of the string; **\n** matches a new-line.

– Within brackets the minus means *through*. For example, **[a–z]** is equivalent to **[abcd...xyz]**. The – can appear as itself only if used as the first or last character. For example, the character class expression **[ ]–]** matches the characters **]** and **–**.

\+ A regular expression followed by + means *one or more times*. For example, **[0–9]+** is equivalent to **[0–9] [0–9]***.

{m} {m,} {m,u}
Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(. . .)$*n*

The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
    . . .
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
    . . .
name = regcmp("([A–Za–z][A–za–z0–9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

This example will match through the string ``Testing3'' and will return the address of the character after the last matched character (the ``4''). The string ``Testing3'' will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
    . . .
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** [see *regcmp*(1)] against *string*.

These routines are kept in **/usr/lib/libPW.a.**

SEE ALSO

regcmp(1), malloc(3C).
ed(1) in the *User's Reference Manual*.

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

NAME

re_comp, re_exec – regular expression handler

SYNOPSIS

char *re_comp(s)
char *s;

re_exec(s)
char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. *Re_exec* checks the argument string against the last string passed to *re_comp*.

Re_comp returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If *re_comp* is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and −1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both *re_comp* and *re_exec* may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for *ed*(1), given the above difference.

DIAGNOSTICS

Re_exec returns −1 for an internal error.

Re_comp returns one of the following strings if an error occurs:

No previous regular expression
Regular expression too long
unmatched \(
missing]
too many \(\) pairs
unmatched \)

SEE ALSO

ed(1), ex(1), egrep(1), fgrep(1), grep(1)

NAME

registerinethost – allocate internet address for workstation

SYNOPSIS

#include <netinet/in.h>

int registerinethost(name, network, netmask, inaddr, aliases)
char *name, *network, *netmask;
struct in_addr *inaddr;
char *aliases;

DESCRIPTION

Registerinethost sends an internet address allocation request to *registrar(1M)* on yp master via the *yp_update(3R)* call. This routine should be used only when yellow page service is enabled in the network.

The arguments for the routine are:

name The host name to be registered. This name must be unique in the yp domain.

network
The internet network number to be used in the allocation. If the *netmask* is supplied, this argument should be an internet address so that the *netmask* can be applied on.

netmask
The internet netmask. If this argument is not used, i.e. it is NULL, the parameter *network* should be a valid internet network number, e.g. *<192.26.61>*. If the argument is used, it must be in the internet standard "." notation, e.g. *<255.255.255.128>* and the *network* parameter must also be a valid internet address, e.g. *<192.26.61.128>*.

inaddr The address of resulting internet address. A NULL indicates the resulting address is not to be returned.

aliases The aliases, each separated by spaces, of the host. The maximum number of characters allowed in the string is MAX_ALIASES defined in <sun/hostreg.h>. A NULL indicates there will be no aliases for the host.

Registerinethost returns NULL on successful registration. The unsuccessful return code are defined in <rpcsvc/ypclnt.h>.

Registerinethost always wait until yellow page data base are pushed to all slave servers.

SEE ALSO

registrar(1M), yp_update(1M), renamehost(3N), unregisterhost(3N), yppush(1M)

AUTHOR

Steve Sun

NAME

remove – remove a file

SYNOPSIS

#include <stdio.h>

int remove (const char *filename);

DESCRIPTION

remove causes the named file to no longer be accessible by the name *filename*. If the file is open, further accesses are permitted, but upon last close the file will be removed from the file system.

SEE ALSO

unlink(2).

DIAGNOSTICS

remove returns zero if it succeeds, –1 otherwise. See *unlink*(2) for possible errors.

NAME

renamehost – rename the existing hostname in yp hosts data base

SYNOPSIS

int renamehost(oldname, newname, aliases, passwd)
char *oldname, *newname, *aliases, *passwd;

DESCRIPTION

Renamehost sends an host rename request to *registrar(1M)* on yp master via the *yp_update(3R)* call. The result is that the new host name will be associated with the original internet address. This routine should be used only when yellow page service is enabled in the network. This function call can not only change the hostname, but also modify the aliases.

The arguments for the routine are:

oldname
The original host name.

newname
The new host name for the internet address. This new name should not be already used by other host. However, it can be the alias of the original host. User may want to use this call to swap the alias and host name. User also can use this call just to modify the alias when *newname* is the same as *oldname*.

aliases The new aliases, each separated by spaces, of the host. The maximum number of characters allowed in the string is MAX_ALIASES defined in <sun/hostreg.h>. The new aliases can be the same as the original aliases, or contains the old host name. A NULL indicates the alias is omitted.

passwd The root password of yp master. If yp master does not have root password, simply pass a NULL.

On successful completion, **Renamehost** returns NULL. The unsuccessful return code are defined in <rpcsvc/ypclnt.h>.

Renamehost always wait until yellow page data base are pushed to all slave servers.

SEE ALSO

registrar(1M), yp_update(1M), registerinethost(3N), unregisterhost(3N), yppush(1M)

NAME

res_query, res_search, res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query (char *dname, int class, int type,
               u_char *answer, int anslen);

int res_search (char *dname, int class, int type,
               u_char *answer, int anslen);

int res_mkquery (int op, char *dname, int class, int type,
               char *data, int datalen, struct rrec *newrr,
               char *buf, int buflen);

int res_send (char *msg, int msglen, char *answer, int anslen);

int res_init (void);

int dn_comp (char *exp_dn, char *comp_dn, int length,
               char **dnptrs, char **lastdnptr);

int dn_expand (char *msg, char *eomorig, char *comp_dn,
               char *exp_dn, int length);
```

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver routines is kept in the structure *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are stored as a simple bit mask containing the bitwise ''or'' of the options enabled.

RES_INIT
: True if the initial name server address and default domain name are initialized (i.e., *res_init* has been called).

RES_DEBUG
: Print debugging messages.

RES_AAONLY
: Accept authoritative answers only. With this option, *res_send* should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
: Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
: Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
: Set the recursion-desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, *res_search* will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH
: If this option is set, *res_search* will search for host names in the current domain and in parent domains; see *hostname*(5). This is used by the standard host lookup routine *gethostbyname*(3N). This option is enabled by default.

The *res_init* routine reads the configuration file (if any; see *resolver*(4)) to get the default domain name, search list and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable LOCALDOMAIN. Initialization normally occurs on the first call to one of the following routines.

The *res_query* function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname* . The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The *res_search* routine makes a query and awaits a response like *res_query*, but in addition, it implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

The remaining routines are lower-level routines used by *res_query*. The *res_mkquery* function constructs a standard query message and places it in *buf*. It returns the size of the query, or −1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in *<arpa/nameser.h>*. The domain name for the query is given by *dname*. *Newrr* is currently unused but is intended for making update messages.

The *res_send* routine sends a pre-formatted query and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or −1 if there were errors.

The *dn_comp* function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or −1 if there were errors. The size of the array pointed to by *comp_dn* is given by *length*. The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of *dn_comp* is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

The *dn_expand* entry expands the compressed domain name *comp_dn* to a full domain name The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or −1 if there was an error.

FILES

/usr/etc/resolv.conf see resolver(4)

SEE ALSO

named(1M), gethostbyname(3N), resolver(4), hostname(5),
RFC1032, RFC1033, RFC1034, RFC1035, RFC974,
The BIND Name Server chapter in the *Network Communications Guide*.

NAME

rexec – return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
int inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call ``getservbyname("exec", "tcp")'' (see *getservent*(3N)) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in *rexecd*(1M).

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3N), rexecd(1M)

NAME

rpc – Remote Procedure Call (RPC) library routines

SYNOPSIS AND DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

#include <rpc/rpc.h>

void
auth_destroy(auth)
  AUTH *auth;

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy()**.

AUTH *
authnone_create()

Create and returns an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
  char *host;
  int uid, gid, len, *aup.gids;

Create and return an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

AUTH *
authunix_create_default()

Calls **authunix_create()** with the appropriate parameters.

callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of **enum clnt_stat** cast to an integer if it fails. The routine **clnt_perrno()** is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create()** for restrictions. You do not have control of timeouts or authentication using this routine.

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in,
outproc, out, eachresult)
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;

Like **callrpc()**, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls **eachresult()**, whose form is:

```
eachresult(out, addr)
        char *out;
        struct sockaddr_in *addr;
```

where *out* is the same as *out* passed to **clnt_broadcast()**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult()** returns zero, **clnt_broadcast()** waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval tout;

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create( )**. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

clnt_destroy(clnt)
CLIENT *clnt;

A macro that destroys the client's RPC handle. Destruction usu ally involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy( )**. If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

CLIENT *
clnt_create(host, prog, vers, proto)
char *host;
u_long prog, vers;
char *proto;

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using **clnt_control( )**.

Warning: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

bool_t
clnt_control(cl, req, info)
CLIENT *cl;
void *info;

A macro used to change or retrieve various information about a client object. *req* is a value to indicate the type of operation, and *info* is a pointer to a buffer to obtain or store the information. For both UDP and TCP, the supported values of *req* are:

To get or set the total timeout, use CLGET_TIMEOUT or CLSET_TIMEOUT with the address of a *struct timeval* variable. For example,

clnt_control(cl, CLGET_TIMEOUT, &tv)

Note: if you set the timeout using **clnt_control()**, the timeout parameter passed to **clnt_call()** will be ignored in all future calls.

To get the server's address, use CLGET_SERVER_ADDR with the address of a *struct sockaddr_in* variable.

The following operations are valid for UDP only:

To get or set the retry timeout, use CLGET_RETRY_TIMEOUT or CLSET_RETRY_TIMEOUT with with the address of a *struct timeval* variable. The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns one if the results were successfully freed, and zero otherwise.

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;

A macro that copies the error structure out of the client handle to the structure at address *errp*.

void
clnt_pcreateerror(s)
char *s;

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. Used when a **clnt_create()**, **clntraw_create()**, **clnttcp_create()**, or **clntudp_create()** call fails.

void
clnt_perrno(stat)
enum clnt_stat stat;

Print a message to standard error corresponding to the condition indicated by *stat*. Used after **callrpc()**.

clnt_perror(clnt, s)
CLIENT *clnt;
char *s;

Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. Used after **clnt_call()**.

char *
clnt_spcreateerror(s)
char *s;

Like **clnt_pcreateerror()**, except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each call.

char *
clnt_sperrno(stat)
enum clnt_stat stat;

Take the same arguments as **clnt_perrno()**, but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message. The string ends with a NEWLINE.

clnt_sperrno() is used instead of **clnt_perrno()** if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**, or if a message

format different than that supported by **clnt_perrno()** is to be used. Note: unlike **clnt_sperror()** and **clnt_spcreaterror()**, **clnt_sperrno()** returns pointer to static data, but the result will not get overwritten on each call.

char *
clnt_sperror(rpch, s)
CLIENT *rpch;
char *s;

Like **clnt_perror()**, except that (like **clnt_sperrno()**) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

CLIENT *
clntraw_create(prognum, versnum)
u_long prognum, versnum;

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create()**. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address **addr*. If **addr->sin_port** is zero, then it is set to the actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O , the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of zero choose suitable defaults. This routine returns NULL if it fails.

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses use UDP/IP as a transport. The remote program is located at Internet address *addr*. If **addr->sin_port** is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of **wait** time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call()**.

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

CLIENT *
clntudp_bufcreate(addr, prognum, versnum, wait, sockp, sendsize, recosize)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
unsigned int sendsize, recosize;

This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses use UDP/IP as a transport. The remote program is located at Internet address *addr*. If **addr->sin_port** is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **sockp**. The UDP transport resends the call message in intervals of **wait** time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call()**.

This allows the user to specify the maximun packet size for sending and receiving UDP-based RPC messages.

void
get_myaddress(addr)
struct sockaddr_in *addr;

Stuff the machine's IP address into **addr*, without consulting the library routines that deal with **/etc/hosts**. The port number is always set to **htons(PMAPPORT)**.

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;

A user interface to the **portmap** service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL . The command '**rpcinfo −p**' uses this routine.

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum, versnum, protocol;

A user interface to the **portmap** service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote **portmap** service. In the latter case, the global variable **rpc_createerr()** contains the RPC status.

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in,
outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
u_long *portp;

A user interface to the **portmap** service, which instructs **portmap** on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc()** and **clnt_call()**. This procedure should be used for a "ping" and nothing else. See also **clnt_broadcast()**.

pmap_set(prognum, versnum, protocol, port)
u_long prognum, versnum, protocol;
u_short port;

A user interface to the **portmap** service, which establishes a mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the machine's **portmap** service. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. This routine returns one if it succeeds, zero otherwise. Automatically done by **svc_register()**.

pmap_unset(prognum, versnum)
u_long prognum, versnum;

A user interface to the **portmap** service, which destroys all mapping between the triple [*prognum*,*versnum*,*] and **ports** on the machine's **portmap** service. This routine returns one if it succeeds, zero otherwise.

registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum, versnum, procnum;
char *(*procname) ();
xdrproc_t inproc, outproc;

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *progname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create()** for restrictions.

struct rpc_createerr rpc_createerr;

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror()** to print the reason why.

svc_destroy(xprt)
SVCXPRT *xprt;

A macro that destroys the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

fd_set svc_fdset;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the **select** system call. This is only of interest if a service implementor does not call **svc_run()**, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to **select**!), yet it may change after calls to **svc_getreqset()** or any creation routines.

int svc_fds;

Similar to **svc_fedset()**, but limited to 32 descriptors. This interface is obsoleted by **svc_fdset()**.

svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs( )**. This routine returns 1 if the results were successfully freed, and zero otherwise.

svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

svc_getreqset(rdfds)
fd_set *rdfds;

This routine is only of interest if a service implementor does not call **svc_run( )**, but instead implements custom asynchronous event processing. It is called when the **select** system call has determined that an RPC request has arrived on some RPC socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

svc_getreq(rdfds)
int rdfds;

Similar to **svc_getreqset( )**, but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset( )**.

svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum, versnum;
void (*dispatch) ();
u_long protocol;

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the **portmap** service. If *protocol* is non-zero, then a mapping of the triple [*prognum*,*versnum*,*protocol*] to **xprt–>xp_port** is established with the local **portmap** service (generally *protocol* is zero, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure *dispatch* has the following form:
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;

The **svc_register( )** routine returns one if it succeeds, and zero otherwise.

svc_run( )

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq( )** when one arrives. This procedure is usually waiting for a **select( )** system call to return.

svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

void
svc_unregister(prognum, versnum)
u_long prognum, versnum;

Remove all mapping of the double [*prognum*,*versnum*] to dispatch routines, and of the triple [*prognum*,*versnum*,*] to port number.

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void
svcerr_decode(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that cannot successfully decode its parameters. See also **svc_getargs( )**.

void
svcerr_noproc(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void
svcerr_noprog(xprt)
SVCXPRT *xprt;

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_progvers(xprt)
SVCXPRT *xprt;

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void
svcerr_weakauth(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK).**

SVCXPRT *
svcraw_create()

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create()**. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size, recv_buf_size;

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, **xprt->xp_sock** is the transport's socket descriptor, and **xprt->xp_port** is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O , users may specify the size of buffers; values of zero choose suitable defaults.

SVCXPRT *
svcfd_create(fd, sendsize, recvsize)
int fd;
u_int sendsize, recvsize;

Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

SVCXPRT *
svcudp_bufcreate(sock, sendsize, recosize)
int sock;

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK** , in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, **xprt−>xp_sock** is the transport's socket descriptor, and **xprt−>xp_port** is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximun packet size for sending and receiving UDP-based RPC messages.

xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;

Used for describing parameters to various **portmap** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

void
xprt_register(xprt)
SVCXPRT *xprt;

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable **svc_fds( )**. Service implementors usually do not need this routine.

void
xprt_unregister(xprt)
SVCXPRT *xprt;

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable **svc_fds()**. Service implementors usually do not need this routine.

SEE ALSO

xdr(3R)
The following chapters in the *Network Communications Guide*: *Remote Procedure Calls: Protocol Specification*, *Remote Procedure Call Programming Guide*, *rpcgen Programming Guide*.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

SysV:

```
#include <sys/types.h>
#include <dirent.h>

int scandir(const char *dirname, struct dirent **namelist[],
            int (*select)(struct dirent *),
            int (*compar)(struct dirent **, struct dirent **));

int alphasort(struct dirent **d1, struct dirent **d2);
```

BSD:

```
#include <sys/types.h>
#include <sys/dir.h>

int scandir(const char *dirname, struct direct **namelist[],
            int (*select)(), int (*compar)());

int alphasort(struct direct **d1, struct direct **d2);
```

DESCRIPTION

The inclusion of *<dirent.h>* selects the System V versions of these routines. For the 4.3BSD versions, include *<sys/dir.h>*.

Scandir reads the directory *dirname* and builds an array of pointers to directory entries using *malloc*(3). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user-supplied subroutine which is called by *scandir* to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to *qsort*(3) to sort the completed array. If this pointer is null, the array is not sorted. *alphasort* is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* (see *malloc*(3)) by freeing each pointer in the array and the array itself.

SEE ALSO

directory(3C), malloc(3), qsort(3C), dirent(4)

DIAGNOSTICS

Returns −1 if the directory cannot be opened for reading or if *malloc*(3) cannot allocate enough memory to hold all the data structures.

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

#include <stdio.h>

int scanf (const char *format, ...);

int fscanf (FILE *stream, const char *format, ...);

int sscanf (const char *s, const char *format, ...);

DESCRIPTION

scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments (represented in the synopsis above by ...) indicating where the converted input should be stored. The results are undefined in there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

i an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading ''0'' implies octal; a leading ''0x'' implies hexadecimal; otherwise, decimal.

n stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an **e**, followed by an optional +, –, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x** and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int n ; float x; char name[50];
n = scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E–1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0** . Or:

```
int i, j; float x; char name[50];
(void) scanf ("%i%2d%f%*d %[0–9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

will assign **9** to *j*, **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* [see *getc*(3S)] will return **a**. Or:

```
int i, j, s, e; char name[50];
(void) scanf ("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* - *s* = 2 . The next call to *getchar* [see *getc*(3S)] will return a blank.

SEE ALSO

getc(3S), printf(3S), stdio(3S), strtod(3C), strtol(3C).

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

CAVEATS

Trailing white space (including a new-line) is left unread unless matched in the control string.

NAME

setbuf, setvbuf, setbuffer, setlinebuf – assign buffering to a stream

SYNOPSIS

#include <stdio.h>

void setbuf (FILE *stream, char *buf);

int setvbuf (FILE *stream, char *buf, int type, size_t size);

int setbuffer (FILE *stream, char *buf, int size);

int setlinebuf (FILE *stream);

DESCRIPTION

The three types of buffering available are unbuffered, fully buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is fully buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. *Fflush*(3S) may be used to force the block out early. By default, output to a terminal is line buffered and all other input/output is fully buffered.

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

    char buf[BUFSIZ];

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in **<stdio.h>**) are:

_IOFBF causes input/output to be fully buffered.

_IOLBF causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

_IONBF causes input/output to be completely unbuffered.

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

Setbuffer and *setlinebuf* are provided for compatibility with 4.3BSD. *Setbuffer*, an alternate form of *setbuf*, is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

Setlinebuf is used to change *stdout* or *stderr* from fully buffered or unbuffered to line buffered. Unlike the other routines, it can be used at any time that the file descriptor is active.

SEE ALSO

fopen(3S), fflush(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* and *setbuffer* return a non-zero value. Otherwise, the value returned will be zero.

NOTES

A common source of error is allocating buffer space as an ''automatic'' variable in a code block, and then failing to close the stream in the same block.

NAME

seteuid, setruid, setegid, setrgid – set user and group IDs

SYNOPSIS

seteuid(euid)
setruid(ruid)
int euid, ruid;

setegid(egid)
setrgid(rgid)
int egid, rgid;

DESCRIPTION

Seteuid (*setegid*) sets the effective user ID (group ID) of the current process.

Setruid (*setrgid*) sets the real user ID (group ID) of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

DIAGNOSTICS

Zero is returned if the user (group) ID is set. Otherwise –1 is returned and *errno* is set to EPERM if the caller is not the super-user and an disallowed ID is used.

SEE ALSO

setreuid(2), setregid(2), setuid(2), setgid(2), getuid(2), getgid(2)

NAME

sethostresorder – specify order of host-address resolution services

SYNOPSIS

int sethostresorder (const char *order);

DESCRIPTION

The *gethostbyname*(3N) and *gethostbyaddr*(3N) routines can access three types of host-address databases:

- the hosts file, /etc/hosts,
- Yellow Pages (YP) and
- the Berkeley Internet Name Domain service ("BIND name server").

sethostresorder allows a program to specify the order of services to resolve Internet addresses and hostnames from these databases.

sethostresorder should be called before the first time *gethostbyname* and *gethostbyaddr* are called. The *order* argument is a character string that contains keywords for the lookup services. See the description of *hostresorder* in *resolver*(4) for the list and meaning of keywords and separators. The colon (:) character is equivalent to white space as a keyword separator. For example,

```
sethostresorder("yp bind local");
sethostresorder("yp:bind:local");
```

are equivalent.

There are two versions of this routine: the standard version in libc and the Yellow Pages version in libsun. The programmatic interface of both versions is identical, except the standard version ignores the YP keyword. The libc default order is ''bind / local'' and the libsun default is ''yp / bind / local''.

This routine overrides the order specified by the *hostresorder* keyword in */usr/etc/resolv.conf* and the HOSTRESORDER environment variable.

DIAGNOSTICS

sethostresorder returns 0 if the order was changed, otherwise it returns –1. Unrecognized keywords are ignored.

SEE ALSO

intro(3), gethostbyname(3N), resolver(4)

NAME

setjmp, longjmp, sigsetjmp, siglongjmp, _setjmp, _longjmp – non-local gotos

SYNOPSIS

#include <setjmp.h>

SysV:

int setjmp (jmp_buf env);

void longjmp (jmp_buf env, int val);

POSIX:

int sigsetjmp (sigjmp_buf, int savemask);

void siglongjmp (sigjmp_buf env, int val);

BSD:

int setjmp (jmp_buf env);

int longjmp (jmp_buf env, int val);

int _setjmp (jmp_buf env);

int _longjmp (jmp_buf env, int val);

To use the BSD versions of *setjmp* and *longjmp*, you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<setjmp.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

All varieties of *setjmp* save their stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by all varieties of *longjmp*. If the return is from a direct invocation, all *setjmps* return the value 0. If the return is from a call to any of the *longjmps*, all *setjmp* routines return a nonzero value.

All *longjmps* restore the environment saved by the last call of *setjmp* with the corresponding *env* argument. After the *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *longjmps* cannot cause *setjmps* to return the value 0. If a *longjmp* is invoked with a second argument of 0, all versions of *setjmp* will return 1. At the time of the second return from a *setjmp*, all accessible data have values as of the time *longjmp* is called. However, global variables will

have the expected values, i.e., those as of the time of the *longjmp* (see example).

SYSV-POSIX-BSD DIFFERENCES

The System V *setjmp/longjmp* perform identically to the 4.3BSD *_setjmp/_longjmp,* i.e., they manipulate only the C stack and registers. The 4.3BSD *setjmp/longjmp* also manipulate the C stack and registers, but additionally save and restore the process's signal mask (see *sigprocmask*(2), *sigblock*(3B), or *sigsetmask*(3B)). The POSIX *sigsetjmp/siglongjmp* calls may act in either manner: the C stack and registers are always saved and restored, but if the *savemask* parameter to *sigsetjmp* is non-zero, the signal mask is saved, and a bit in *env* is set to indicate that it was saved. *siglongjmp* checks that bit to determine if it should restore the mask or not.

Note that the System V *longjmp* and POSIX *siglongjmp* return *void,* whereas the 4.3BSD *longjmp* and *_longjmp* return an integer.

EXAMPLE

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
    if (setjmp(env) != 0) {
        (void) printf("2nd return from setjmp: i = %d\n", i);
        exit(0);
    }
    (void) printf("1st return from setjmp: i = %d\n", i);
    i = 1;
    g();
    /*NOTREACHED*/
}

g()
{
    longjmp(env, 1);
    /*NOTREACHED*/
}
```

The program's output is:

```
1st return from setjmp: i = 0
2nd return from setjmp: i = 1
```

SEE ALSO

sigaction(2), sigprocmask(2), signal(2), sigblock(3B), sigsetmask(3B), sigvec(3B), signal(3B).

WARNINGS

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, absolute chaos is guaranteed.

If different versions of these jmp routines are mixed, unpredictable signal masking may occur.

BUGS

The values of the registers on the second return from the *setjmps* are the register values at the time of the first call to *setjmp*, not those at the time of the *longjmp*. This means that variables in a given function may behave differently in the presence of *setjmp*, depending on whether they are register or stack variables.

NAME

gethostsex – get the byte sex of the host machine

swap_*() - swap the sex of the specified structure

SYNOPSIS

```
#include <sex.h>
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <symconst.h>
#include <cmplrs/stsupport.h>
#include <reloc.h>
#include <ar.h>

int gethostsex()

long swap_word(word)
long word;

short swap_half(half)
short half;

void swap_filehdr(pfilehdr, destsex)
FILHDR *pfilehdr;
long destsex;

void swap_aouthdr(paouthdr, destsex)
AOUTHDR *paouthdr;
long destsex;

void swap_scnhdr(pscnhdr, destsex)
SCNHDR *pscnhdr;
long destsex;

void swap_hdr(phdr, destsex)
pHDRR phdr;
long destsex;

void swap_fd(pfd, count, destsex)
pFDR pfd;
long count;
long destsex;

void swap_fi(pfi, count, destsex)
pFIT pfi;
long count;
long destsex;
```

void swap_sym(psym, count, destsex)
pSYMR psym;
long count;
long destsex;

void swap_ext(pext, count, destsex)
pEXTR pext;
long count;
long destsex;

void swap_pd(ppd, count, destsex)
pPDR ppd;
long count;
long destsex;

void swap_dn(pdn, count, destsex)
pRNDXR pdn;
long count;
long destsex;

void swap_opt(popt, count, destsex)
pOPTR popt;
long count;
long destsex;

void swap_aux(paux, type, destsex)
pAUXU paux;
long type;
long destsex;

void swap_reloc(preloc, count, destsex)
struct reloc *preloc;
long count;
long destsex;

void swap_ranlib(pranlib, count, destsex)
struct ranlib *pranlib;
long count;
long destsex;

DESCRIPTION

To use these routines, the library *libmld.a* must be loaded.

Gethostsex returns one of two constants BIGENDIAN or LITTLEENDIAN for the sex of the host machine. These constants are in *sex.h*.

All *swap_** routines that swap headers take a pointer to a header structure to change the byte's sex. The *destsex* argument lets the swap routines decide whether to swap bitfields before or after swapping the words they occur in. If *destsex* equals the hostsex of the machine you are running on, the flip

happens before the swap; otherwise, the flip happens after the swap. Although not all routines swap structures containing bitfields, the destsex is required in the anticipation of future need.

The *swap_aux* routine takes a pointer to an aux entry and a *type*, which is a ST_AUX_* constant in cmplrs/stsupport.h. The constant specifies the type of the aux entry to change the sex of. All other *swap_** routines are passed a pointer to an array of structures and a *count* of structures to change the byte sex of. The routines *swap_word* and *swap_half* are macros declared in *sex.h.* Only the include files necessary to describe the structures being swapped need be included.

AUTHOR

Kevin Enderby

NAME

sigblock – block signals from delivery to process (4.3BSD)

SYNOPSIS

#include <signal.h>

int sigblock(int mask);

mask = sigmask(int signum);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<signal.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in *mask* is a 1 (numbering the bits from 1 to 32); the macro *sigmask* is provided to construct the mask for a given *signum*.

It is not possible to block **SIGKILL**, **SIGSTOP**, or **SIGCONT**; this restriction is imposed silently by the system.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(3B), sigvec(3B), sigsetmask(3B)

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

signal – simplified software signal facilities (4.3BSD)

SYNOPSIS

#include <signal.h>

int (*signal(int sig, int (*func)(int, ...)))(int, ...);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<signal.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

signal is a simplified interface to the more general *sigvec*(3B) facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *termio*(7)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the **SIGKILL** and **SIGSTOP** signals, the *signal* call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all legal signals (as defined in *<sys/signal.h>*):

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it

SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

Func is assigned one of three values: **SIG_DFL** or **SIG_IGN**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext* (defined in <*sys/signal.h*>) that contains the processor context at the time of the signal.

The signal-catching function remains installed after it is invoked. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted. See WARNINGS below.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD**, **SIGWINCH**, **SIGPWR**, **SIGURG**, and **SIGIO**, the handler parameter is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

SIG_DFL – ignore signal
The signal is to be ignored.

SIG_IGN – ignore signal
The signal is to be ignored.

function address – catch signal
If the signal is **SIGPWR**, **SIGWINCH**, **SIGURG**, or **SIGIO**, the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. Unlike *signal*(2) and *sigset*(2), re-attaching the handler before exiting it will NOT ensure that no **SIGCLD**'s will be missed. See *wait*(2) for an example of parent code waiting on children.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL.**

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO**. **SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the controlling terminal.

NOTES

If *func* is **SIG_DFL**, the default action for signal *sig* is reinstated. If *func* is **SIG_IGN** the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike the System V signal routine, the handler *func* **remains installed** after a signal has been delivered.

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (typically <Ctrl>-Z) will not act upon any delivered signals other than **SIGKILL** until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e., a non-job-control signal which they are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked

(a call to *unblockproc*(2) or *unblockprocall*(2) is necessary).

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of pending signals.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored, the blocked signal mask is unchanged and pending signals remain pending.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

signal will fail and no action will take place if one of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for **SIGKILL** or **SIGSTOP**.

[EINVAL] An attempt is made to ignore **SIGCONT** (by default **SIGCONT** is ignored).

SEE ALSO

kill(3B), sigvec(3B), sigblock(3B), sigsetmask(3B), sigpause(3B), setjmp(3), blockproc(2).

CAVEATS (IRIX)

4.2BSD attempts to restart system calls which are interrupted by signal receipt; 4.3BSD gives the programmer a choice of restart or failed-return-with-error via the **SV_INTERRUPT** flag in *sigvec* or use of the *siginterrupt* library routine. IRIX provides *only* the fail-with-error option. The affected system calls are *read*(2), *write*(2), *open*(2), *ioctl*(2), and *wait*(2). Refer to the *sigset*(2) man page for a more detailed description of the behavior.

Because 4.3BSD and System V both have *signal* system calls, programs using 4.3BSD's version are actually executing *BSDsignal*. This is transparent to the programmer except when attempting to set breakpoints in **dbx**; the breakpoint must be set at *BSDsignal*.

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sigpause – atomically release blocked signals and wait for interrupt (4.3BSD)

SYNOPSIS

#include <signal.h>

int sigpause(int mask);

mask = sigmask(int signum);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including *<signal.h>*, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

sigpause assigns *mask* to the set of masked signals and then waits for a signal to arrive; upon return the original set of masked signals is restored after executing the handler(s) (if any) installed for the awakening signal(s). *mask* is usually 0 to indicate that no signals are now to be blocked. The macro *sigmask* is provided to construct the mask for a given signal number. *Sigpause* always terminates by being interrupted, returning –1 with the global integer *errno* set to EINTR.

In normal usage, a signal is blocked using *sigblock*(3B), to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigpause* with the mask returned by *sigblock*.

SEE ALSO

sigblock(3B), sigvec(3B)

CAVEATS (IRIX)

Because 4.3BSD and System V both have *sigpause* system calls, programs using 4.3BSD's version are actually executing *BSDsigpause*. This is transparent to the programmer except when attempting to set breakpoints in **dbx**; the breakpoint must be set at *BSDsigpause*.

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sigsetmask – set current signal mask (4.3BSD)

SYNOPSIS

#include <signal.h>

int sigsetmask(int mask);

mask = sigmask(int signum);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define **_BSD_SIGNALS** or **_BSD_COMPAT** before including <*signal.h*>, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in *mask* is a 1 (numbering the bits from 1 to 32); the macro *sigmask* is provided to construct the mask for a given *signum*.

The system quietly disallows **SIGKILL**, **SIGSTOP**, or **SIGCONT** to be blocked.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(3B), sigvec(3B), sigblock(3B), sigpause(3B)

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sigsetops: sigaddset, sigdelset, sigemptyset, sigfillset, sigismember, sgi_altersigs, sgi_sigffset, sgi_siganyset, sgi_dumpset – signal set manipulation and examination routines (POSIX, with SGI-specific additions)

SYNOPSIS

POSIX

#include <signal.h>

int sigaddset(sigset_t *set, int sig);

int sigdelset(sigset_t *set, int sig);

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigismember(sigset_t *set, int sig);

SGI

int sgi_altersigs(int action, sigset_t *set, int sigarray[]);

int sgi_sigffset(sigset_t *set, int clearit);

int sgi_siganyset(sigset_t *set);

int sgi_dumpset(sigset_t *set);

DESCRIPTION

These library calls modify or return information about the disposition of the signal mask pointed to by *set.* The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A global *signal mask* defines the set of signals currently blocked from delivery to a process; it may be changed with a *sigprocmask*(2) call. The masks submitted as parameters to *sigprocmask*, *sigaction*, and *sigsuspend* and returned by *sigpending* may be constructed, altered, and examined via the sigsetops described in this man page. They do *NOT* themselves alter the global signal mask. The masks that the routines manipulate are of type *sigset_t.*

sigaddset adds *sig* to the specified set.

sigdelset deletes *sig* from the specified set.

sigemptyset clears all signals in the specified set.

sigfillset sets all signals in the specified set.

sigismember returns 1 if *sig* is a member of the specified set, else returns 0.

SGI-SPECIFIC FUNCTIONS

The following four functions, although *not* part of the POSIX specification, provide additional capabilities:

sgi_altersigs performs *action* on the specified signal set, for each signal in *sigarray*. *Action* may be ADDSIGS or DELSIGS (defined in *<sys/signal.h>*). The final signal entry in *sigarray* must be followed by a 0 entry (in this way *sgi_altersigs* knows how many signals to process). The array may include all legal signals; however, if the intent is to set or clear all signals the *sigaddset* and *sigdelset* routines are more efficient. Any illegal signal numbers are silently skipped. *sgi_altersigs* returns the number of signals which were processed, or −1 with *errno* set to **[EINVAL]** if *action* is not ADDSIGS or DELSIGS.

sgi_sigffset returns the number of the lowest pending signal in *set*. If none are pending, it returns 0. If *clearit* is non-zero, the returned signal is cleared in the mask. In this way *sgi_sigffset* may be used to sequentially examine the signals in a mask without duplication.

sgi_siganyset(set) returns 1 if *any* signals are set in the specified mask, otherwise it returns 0. The mask is not altered.

sgi_dumpset displays the specified set of signals as a bit-vector, primarily for debugging purposes.

The following is a list of all legal signals (defined in *<sys/signal.h>*):

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGABRT	6	abort
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	child status has changed
SIGPWR	19	Power-fail restart
SIGSTOP	20	stop (cannot be caught, blocked, or ignored)

SIGTSTP	21	stop signal generated from keyboard
SIGPOLL	22	Pollable event occurred
SIGIO	23	i/o is possible on a descriptor (see *fcntl*(2))
SIGURG	24	urgent condition present on socket
SIGWINCH	25	window size change
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))
SIGCONT	28	continue after stop (cannot be blocked)
SIGTTIN	29	background read attempted from control terminal
SIGTTOU	30	background write attempted to control terminal
SIGXCPU	31	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	32	file size limit exceeded (see *setrlimit*(2))

ERRORS

In every routine, the *set* parameter is a *pointer* to sigset_t. All of these functions are library routines (executing in user space); therefore if they are passed a *REFERENCE* to *set* instead of a *POINTER*, the compiler will issue a warning, and when the program is run the process will receive a memory fault signal [**SIGSEGV**] and terminate (unless the process has installed a handler for **SIGSEGV**).

All routines which require a *sig* parameter will fail, returning −1 and setting *errno* to [**EINVAL**] if *sig* is not a valid signal number.

SEE ALSO

sigaction(2), sigprocmask(2), sigpending(2), sigsuspend(2), sigsetjmp(3)

NAME

sigvec – 4.3BSD software signal facilities

SYNOPSIS

#include <signal.h>

```
struct sigvec {
        int     (*sv_handler)(int, ...);
        int     sv_mask;
        int     sv_flags;
};
```

int sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);

To use any of the BSD signal routines (*kill*(3B), *killpg*(3B), *sigblock*(3B), *signal*(3B), *sigpause*(3B), *sigsetmask*(3B), *sigvec*(3B)) you must either

1) #define _BSD_SIGNALS or _BSD_COMPAT before including <*signal.h*>, or

2) specify one of them in the compile command or makefile:

```
cc -D_BSD_SIGNALS -o prog prog.c
```

DESCRIPTION

sigvec specifies and reports on the way individual signals are to be handled in the calling process. If *vec* is non-zero, it alters the way the signal will be treated — default behavior, ignored, or handled via a routine — and the signal mask to be used when delivering the signal if a handler is installed. If *ovec* is non-zero, the previous handling information for the signal is returned to the user. In this way (a NULL *vec* and a non-NULL *ovec*) the user can inquire as to the current handling of a signal without changing it. If both *vec* and *ovec* are NULL, *sigvec* will return –1 and set *errno* to EINVAL if *sig* is an invalid signal (else 0), allowing an application to dynamically determine the set of signals supported by the system.

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigblock*(3B) or *sigsetmask*(3B) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a *sigblock* or *sigsetmask* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*'ing in the signal mask associated with the handler to be invoked.

The following is a list of all legal signals (as defined in <*sys/signal.h*>):

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes

SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

sv_handler is assigned one of three values: **SIG_DFL** or **SIG_IGN**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal
The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

function address – catch signal
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext* (defined in <*sys/signal.h*>) that contains the processor context at the time of the signal.

The signal-catching function remains installed after it is invoked. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted. See WARNINGS below.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD, SIGWINCH, SIGPWR, SIGURG,** and **SIGIO,** the handler parameter is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are:

SIG_DFL – ignore signal
The signal is to be ignored.

SIG_IGN – ignore signal
The signal is to be ignored.

function address – catch signal
If the signal is **SIGPWR, SIGWINCH, SIGURG,** or **SIGIO,** the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. Unlike *signal*(2) and *sigset*(2), re-attaching the handler before exiting it will NOT ensure that no **SIGCLD**'s will be missed. See *wait*(2) for an example of parent code waiting on children.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL.**

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be

resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO**. **SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the controlling terminal.

NOTES

Once a signal handler is installed, it remains installed until another *sigvec* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sv_handler* to **SIG_DFL**; this default is termination with a core image for signals marked **[1]**. If *sv_handler* is **SIG_IGN** the signal is subsequently ignored, and pending instances of the signal are discarded.

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (typically <ctrl>-Z) will not act upon any delivered signals other than **SIGKILL** until the job is restarted. Processes which are blocked via a *blockproc*(2) system call will unblock if they receive a signal which is fatal (i.e., a non-job-control signal which they are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (a call to *unblockproc*(2) or *unblockprocall*(2) is necessary).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of the pending signals.

The *exec*(2) routines reset all caught signals to default action and clear all handler masks. Ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

The mask specified in *vec* is not allowed to block **SIGKILL**, **SIGSTOP**, or **SIGCONT**. This is enforced silently by the system.

RETURN VALUE

A 0 value indicated that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

sigvec is a library routine (executing in user space): if either *vec* or *ovec* points to memory that is not a valid part of the process address space, the process will receive a memory fault (**SIGSEGV**) signal and terminate (unless it has installed a handler for **SIGSEGV**). If the invalid pointer is the

result of using a REFERENCE instead of a POINTER, the compiler will issue a warning.

sigvec will fail and no new signal handler will be installed if one of the following occurs:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for **SIGKILL** or **SIGSTOP**.

[EINVAL] An attempt is made to ignore **SIGCONT** (by default **SIGCONT** is ignored).

SEE ALSO

kill(3B), sigblock(3B), sigsetmask(3B), sigpause(3B), sigvec(3B), setjmp(3), blockproc(2).

CAVEATS (IRIX)

4.2BSD attempts to restart system calls which are interrupted by signal receipt; 4.3BSD gives the programmer a choice of restart or failed-return-with-error via the **SV_INTERRUPT** flag in *sigvec* or use of the *siginterrupt* library routine. IRIX provides *only* the fail-with-error option. The affected system calls are *read*(2), *write*(2), *open*(2), *ioctl*(2), and *wait*(2). Refer to the *sigset*(2) man page for more a detailed description of the behavior.

4.3BSD allows exception handlers to execute on separate, user-specified stacks (and provides the *sigstack*(2) system call for the purpose); IRIX's BSD signal routines do *NOT* provide this capability. A call with **SV_ONSTACK** bit (0x0) set in the *sv_flags* field will return −1, with *errno* set to **EINVAL**.

WARNING (IRIX)

The 4.3BSD and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sinh, cosh, tanh, fsinh, fcosh, ftanh – hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh(double x);

float fsinh(float x);

double cosh(double x);

float fcosh(float x);

double tanh(double x);

float ftanh(float x);

DESCRIPTION

These functions compute the designated hyperbolic functions for double and float arguments.

ERROR (due to Roundoff etc.)

Below 2.4 *ulps*; an *ulp* is one *U*nit in the *L*ast *P*lace.

DIAGNOSTICS

sinh and *cosh* return +infinity if the correct value would overflow. *sinh* may return –infinity for negative *x* if the correct value would overflow.

SEE ALSO

math(3M)

AUTHOR

W. Kahan, Kwok–Choi Ng

NAME

sleep – suspend execution for interval

SYNOPSIS

include <unistd.h>

unsigned sleep (uint seconds);

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the ''unslept'' amount (the requested time minus the time actually slept) in case of premature arousal due to a caught signal.

NOTES

Unlike previous implementations, *sleep* is implemented with the *sginap(2)* system call rather than with *alarm(2).* Therefore, there are no unusual side effects with the SIGALRM signal; its effect is like that of any other signal.

SEE ALSO

alarm(2), sginap(2), pause(2), sigaction(2), sigset(2)

NAME

sputl, sgetl – access long integer data in a machine-independent fashion

SYNOPSIS

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;

DESCRIPTION

sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program that uses these functions must be loaded with the object-file access routine library **libmld.a.**

NAME

sqrt, fsqrt, cbrt – cube root, square root

SYNOPSIS

#include <math.h>

double sqrt(double x);

float fsqrt(float x);

double cbrt(double x);

DESCRIPTION

sqrt(x) and *fsqrt(x)* return the square root of x for double and float data types respectively.

cbrt(x) returns the cube root of x.

sqrt(x) and *fsqrt(x)* are available in **libm.a**, **libm43.a**, and **libfastm.a**. The first two libraries are described in *math*(3m).

libfastm.a contains only very fast versions *sqrt(x)* and *fsqrt(x)* which are slightly less accurate than the **libm.a** versions.

DIAGNOSTICS

sqrt returns the default quiet NaN when *x* is negative indicating invalid operation.

ERROR (due to Roundoff etc.)

cbrt is accurate to within 0.7 *ulp*s.

libm.a *sqrt* on MIPS machines conforms to IEEE 754 and is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round to nearest).

The **libfastm.a** *sqrt* and *fsqrt* error is one *ulp*.

An *ulp* is one *U*nit in the *L*ast *P*lace carried.

SEE ALSO

math(3M)

AUTHOR

W. Kahan

NAME

ssignal, gsignal – software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 16. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(2), sigset(2).

NOTES

There are some additional signals with numbers outside the range 1 through 16 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 16 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

staux – routines that provide scalar interfaces to auxiliaries

SYNOPSIS

#include <syms.h>

long st_auxbtadd(bt)
long bt;

long st_auxbtsize(iaux,width)
long iaux;
long width;

long st_auxisymadd (isym)
long isym;

long st_auxrndxadd (rfd,index)
long rfd;
long index;

long st_auxrndxadd (idn)
long idn;

void st_addtq (iaux,tq)
long iaux;
long tq;

long st_tqhigh_aux(iaux)
long iaux;

void st_shifttq (iaux, tq)
int iaux;
int tq;

long st_iaux_copyty (ifd, psym)
long ifd;
pSYMR psym;

void st_changeaux (iaux, aux)
long iaux;
AUXU aux;

void st_changeauxrndx (iaux, rfd, index)
long iaux;
long rfd;
long index;

DESCRIPTION

Auxiliary entries are unions with a fixed length of four bytes per entry. Much information is packed within the auxiliaries. Rather than have the compiler front-ends handle each type of auxiliary entry directly, the

following set of routines provide a high-level scalar interface to the auxiliaries:

st_auxbtadd
: Adds a type information record (TIR) to the auxiliaries. It sets the basic type (bt) to the argument and all other fields to zero. The index to this auxiliary entry is returned.

st_auxbtsize
: Sets the bit in the TIR, pointed to by the *iaux* argument. This argument says the basic type is a bit field and adds an auxiliary with its width in bits.

st_auxisymadd
: Adds an index into the symbol table (or any other scalar) to the auxiliaries. It sets the value to the argument that will occupy all four bytes. The index to this auxiliary entry is returned.

st_auxrndxadd
: Adds a relative index, RNDXR, to the auxiliaries. It sets the rfd and index to their respective arguments. The index to this auxiliary entry is returned.

st_auxrndxadd_idn
: Works the same as *st_auxrndxadd* except that RNDXR is referenced by an index into the dense number table.

st_iaux_copyty
: Copies the type from the specified file (ifd) for the specified symbol into the auxiliary table for the current file. It returns the index to the new aux.

st_shifttq
: Shifts in the specified type qualifier, tq (see sym.h), into the auxiliary entry TIR, which is specified by the 'iaux' index into the current file. The current type qualifiers shift up one tq so that the first tq (tq0) is free for the new entry.

st_addtq
: Adds a type qualifier in the highest or most significant non-tqNil type qualifier.

st_tqhigh_iaux
: Returns the most significant type qualifier given an index into the files aux table.

st_changeaux
: Changes the iauxth aux in the current file's auxiliary table to aux.

st_changeauxrndx
: Converts the relative index (RNDXR) auxiliary, which is specified by iaux, to the specified arguments.

The programs must be loaded with the object file access routine library libmld.a.

AUTHOR

Mark I. Himelstein

SEE ALSO

stfd(3x), syms(4).

BUGS

The interface will added to incrementally, as needed.

NAME

stcu – routines that provide a compilation unit symbol table interface

SYNOPSIS

```
#include <syms.h>

pCHDRR st_cuinit ()

void st_setchdr (pchdr)
pCHDRR        pchdr;

pCHDRR st_currentpchdr()

void st_free()

long st_extadd (iss, value, st, sc, index)
long iss;
long value;
long st;
long sc;
long index;

pEXTR st_pext_iext (iext)
long    iext;

pEXTR st_pext_rndx (rndx)
RNDXR rndx;

long st_iextmax()

long st_extstradd (str)
char *str;

char *st_str_extiss (iss)
long iss;

long st_idn_index_fext (index, fext)
long index;
long fext;

long st_idn_rndx (rndx)
RNDXR rndx;

pRNDXR st_pdn_idn (idn)
long idn;
RNDXR st_rndx_idn (idn)
long idn;

void st_setidn (idndest, idnsrc)
long idndest;
long idnsrc;
```

DESCRIPTION

The *stcu* routines provide an interface to objects that occur once per object rather than once per file descriptor (for example, external symbols, strings, and dense numbers). The routines provide access to the current *chdr* (compile time hdr), which represents the symbol table in running processes with pointers to symbol table sections rather than indices and offsets used in the disk file representation.

A new symbol table can be created with *st_cuinit*. This routine creates and initializes a CHDRR (see *<cmplrs/stsupport.h>*). The CHDRR is the current chdr and is used in all later calls. **NOTE:** A chdr can also be created with the read routines (see *stio*(3x)). The *2st_cuinit* routine returns a pointer to the new CHDRR record.

st_currentchdr
: Returns a pointer to the current chdr.

st_setchdr
: Sets the current chdr to the *pchdr* argument and sets the per file structures to reflect a change in symbol tables.

st_free
: Frees all constituent structures associated with the current chdr.

st_extadd
: Lets you add to the externals table. It returns the index to the new external for future reference and use. The *ifd* field for the external is filled in by the current file (see *stfd.3*). For more details on the parameters, see *<sym.h>*.

st_pext_iext

and *st_pext_rndx* Returns pointers to the external, given a index referencing them. The latter routine requires a relative index where the *index* field should be the index in external symbols and the *rfd* field should be the constant ST_EXTIFD. **NOTE:** The externals contain the same structure as symbols (see the *SYMR* and *EXTR* definitions).

st_iextmax
: Returns the current number of entries in the external symbol table.

The *iss* field in external symbols (the index into string space) must point into external string space.

st_extstradd
: Adds a null-terminated string to the external string space and returns its index.

st_str_extiss
Converts that index into a pointer to the external string.

The dense number table provides a convenience to the code optimizer, generator, and assembler. This table lets them reference symbols from different files and externals with unique densely packed numbers.

st_idn_index_fext
Returns a new dense number table index, given an index into the symbol table of the current file (or if *fext* is set, the externals table).

st_idn_rndx
Returns a new dense number, but expects a RNDXR (see <*sym.h*>) to specify both the file index and the symbol index rather than implying the file index from the current file. The RNDXR contains two fields: an index into the externals table and a file index (*rsyms* can point into the symbol table, as well). The file index is ST_EXTIFD (see <*cmplrs/stsupport.h*>) for externals.

st_rndx_idn
Returns a RNDX, given an index into the dense number table.

st_pdn_idn
Returns a pointer to the RNDXR index by the 'idn' argument.

The programs must be loaded with the object file access routine library *libmld.a*.

AUTHOR

Mark I. Himelstein

SEE ALSO

stfe(3x), stfd(3x)

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer-constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* [see *fopen*(3S)] will cause it to

become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *setbuf*(3S) or *setvbuf*() in *setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc: ftok – standard interprocess communication package

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *path, char id);

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns **(key_t) −1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

stfd – routines that provide access to per file descriptor section of the symbol table

SYNOPSIS

#include <syms.h>

long st_currentifd ()

long st_ifdmax ()

void st_setfd (ifd)
long ifd;

long st_fdadd (filename)
char *filename;

long st_symadd (iss, value, st, sc, freloc, index)
long iss;
long value;
long st;
long sc;
long freloc;
long index;

long st_auxadd (aux)
AUXU aux;

long st_stradd (cp)
char *cp;

long st_lineadd (line)
long line;

long st_pdadd (isym)
long isym;

long st_ifd_pcfd (pcfd1)
pCFDR pcfd1;

pCFDR st_pcfd_ifd (ifd)
long ifd;

pSYMR st_psym_ifd_isym (ifd, isym)
long ifd;
long isym;

pAUXU st_paux_ifd_iaux (ifd, iaux)
long ifd;
long iaux;

pAUXU st_paux_iaux (iaux)
long iaux;

char *st_str_iss (iss)
long iss;

char *st_str_ifd_iss (ifd, iss)
long ifd;
long iss;

pPDR st_ppd_ifd_isym (ifd, isym)
long ifd;
long isym;

char * st_malloc (ptr,psize,itemsize,baseitems)
char *ptr;
long *size;
long itemsize;
long baseitems;

DESCRIPTION

The *stfd* routines provide an interface to objects handled on a per file descriptor (or fd) level (for example, local symbols, auxiliaries, local strings, line numbers, optimization entries, procedure descriptor entries, and the file descriptors). These routines constitute a group because they deal with objects corresponding to fields in the *FDR* structure.

A fd can be activated by reading an existing one into memory or by creating a new one. The compilation unit routines *st_readbinary* and *st_readst* read file descriptors and their constituent parts into memory from a symbol table on disk.

St_fdadd adds a file descriptor to the list of file descriptors. The *lang* field is initialized from a user specified global *st_lang* that should be set to a constant designated for the language in *<symconst.h>*. The *fMerge* field is initialized from the user specified global *st_merge* that specifies whether the file is to start with the attribute of being able to be merged with identical files at load time. The *fBigendian* field is initialized by the *gethostsex*(3) routine, which determines the permanent byte ordering for the auxiliary and line number entries for this file.

St_fdadd adds the null string to the new files string table that is accessible by the constant issNull. It also adds the filename to the string table and sets the *rss* field. Finally, the current file is set to the newly added file so that later calls operate on that file.

All routines for fd-level objects handle only the current file unless a file index is specified. The current file can also be set with *st_setfd.*

Programs can find the current file by calling *st_currentifd,* which returns the current index. Programs can find the number of files by calling *st_ifdmax.* The fd routines only require working with indices to do most things. They allow more in-depth manipulation by allowing users to get the compile time file descriptor *(CFDR* see *<cmplrs/stsupport.h>*) that contains memory pointers to the per file tables (rather than indices or offsets used in disk files). Users can retrieve a pointer to the CFDR by calling *st_pcfd_ifd* with the index to the desired file. The inverse mapping *st_ifd_pcfd* exists, as well.

Each of fd's constituent parts has an add routine: *st_symadd, st_stradd, st_lineadd, st_pdadd,* and *st_auxadd.* The parameters of the add routines correspond to the fields of the added object (see *<sym.h>*). The *pdadd* routine lets users fill in the isym field only. Further information can be added by directly accessing the procedure descriptor entry.

The add routines return an index that can be used to retrieve a pointer to part of the desired object with one of the following routines: *st_psym_isym, st_str_iss,* and *st_paux_iaux.* **NOTE:** These routines only return objects within the current file. The following routines allow for file specification: *st_psym_ifd_isym, st_aux_ifd_iaux,* and *st_str_ifd_iss.*

St_ppd_ifd_isym allows access to procedures through the file index for the file where they occur and the isym field of the entry that points at the local symbol for that procedure.

The return index from st_symadd should be used to get a dense number (see stcu). That number should be the ucode block number for the object the symbol describes.

The programs must be loaded with the object file access routine library libmld.a.

AUTHOR

Mark I. Himelstein

SEE ALSO

stfe(3x), stcu(3x)

BUGS

The interface will added to incrementally, as needed.

NAME

stfe – routines that provide a high-level interface to basic functions needed to access and add to the symbol table

SYNOPSIS

```
#include <syms.h>

long st_filebegin (filename)
char *filename;

long st_endallfiles ()

long st_fileend (idn)
long idn;

long st_blockbegin(iss, value, sc)
long iss;
long value;
long sc;

long st_textblock()

long st_blockend(size)
long size;

long st_procend(idn)
long idn

long st_procbegin (idn)
long idn;

char *st_str_idn (idn)
long idn;

char *st_sym_idn (idn,  sc, st, value, index)
long idn;
long *sc;
long *st;
long *value;
long *index;

long st_abs_ifd_index (ifd, index)
long ifd;
long index;

long st_fglobal_idn (idn)
long idn;

pSYMR st_psym_idn_offset (idn, offset)
long idn;
long offset;
```

long st_pdadd_idn (idn)
long idn;

DESCRIPTION

The *stfe* routines provide a high-level interface to the symbol table based on common needs of the compiler front-ends.

st_filebegin

Takes a file name and calls *st_fdadd* (see the *stfd* manual page). If it's a new file, a symbol is added to the symbol table for it and the user supplied routine, *st_feinit,* is called. This allows special once per file things to be initialized (for example, the C front-end adds basic type auxiliaries to each file's aux table so that all variables of that type can refer to a single instance instead of making individual copies of them). *st_filebegin* returns a dense number that references the symbol added for this file. It tracks files as they appear in a CPP line directive with a stack. It detects (from the order of the CPP directives) that a file ends and calls *st_filend.* If a file is closed with a *st_fileend*, a new instance of the filename is created (for example, multiply included files).

st_fileend

Requires the dense number from the corresponding *st_filebegin* call for the file in question. It then generates an end symbol and patches the references so that the index field of the begin file points to that of one beyond the end file. The end file points to the begin file.

st_endallfiles

Is called at the end of execution to close off all files that haven't been ended by previous calls to *st_filebegin.* CPP directives might not reflect the return to the original source file; therefore, this routine can possibly close many files.

st_blockbegin

Supports language blocks (for example, C's left curly brace blocks), beginning of structures, and unions. If the storage class is scText, it is the former; if it is scInfo, it is one of the latter. The iss (index into string space) specifies the name of the structure/etc, if any.

If the storage class is scText, we must check the result of *st_blockbegin.* It returns a dense number for outer blocks and a zero for nested blocks. The non-zero block number should be used in the BGNB ucode. Users of languages without nested blocks that provide variable declarations can ignore the rest of this paragraph. Nested blocks are two-staged: one stage happens when we detect the language block and the other stage happens

when we know the block has content. If the block has content (for example, local variables), the front-end must call *st_textblock* to get a non-zero dense number for the block's BGNB ucode. If the block has no content and *st_textblock* is not called, the block's *st_blockbegin* and *st_blockend* do not produce block and end symbols.

If it is scInfo, *st_blockbegin* creates a begin block symbol in the symbol table and returns a dense number referencing it. The dense number is necessary to build the auxiliary required to reference the structure/etc. It goes in the aux after the TIR along with a file index. This dense number is also noted in a stack of blocks used by *st_blockend*.

st_blockbegin should not be called for language blocks when the front-end is not producing debugging symbols.

st_blockend requires that blocks occur in a nested fashion. It retrieves the dense number for the most recently started block and creates a corresponding end symbol. As in *fileend*, both the begin and end symbol index fields point at the other end's symbol. If the symbol ends a structure/etc., as determined by the storage class of the begin symbol, the size parameter is assigned to the begin symbol's value field. It's usually the size of the structure or max value of a enum. We only know it at this point. The dense number of the end symbol is returned so that the ucode ENDB can be use it. If it is an ignored text block, the dense number is zero and no ENDB should be generated.

In general, defined external procedures or functions appear in the symbols table and the externals table. The external table definition must occur first through the use of a *st_extadd*. After that definition, *st_procbegin* can be called with a dense number referring to the external symbol for that procedure. It checks to be sure we have a defined procedure (by checking the storage class). It adds a procedure symbol to the symbol table. The external's index should point at its auxiliary data type information (or if debugging is off, indexNil). This index is copied into the regular symbol's index field or a copy of its type is generated (if the external is in a different file than the regular symbol). Next, we put the index to symbol in the external's index field. The external's dense number is used as a block number in ucodes referencing it and is used to add a procedure when in the *st_pdadd_idn*.

st_procend

Creates an end symbol and fixes the indices as in *blockend* and *fileend*, except that the end procedure reference is kept in the begin procedure's aux rather than in the index field (because the begin procedure has a type as well as an end reference). This must be called with the dense number of the procedure's external symbol as an argument and returns the dense number of the end

symbol to be used in the END ucode.

st_str_idn
Returns the string associated with symbol or external referenced by the dense number argument. If the symbol was anonymous (for example, there was no symbol) a (char *)-1 is returned.

st_sym_idn
Returns the same result as *st_str_idn*, except that the rest of the fields of the symbol specified by the *idn* are returned in the arguments.

st_fglobal_idn
Returns a 1 if the symbol associated with the specified idn is non-static; otherwise, a 0 is returned.

st_abs_ifd_index
Returns the absolute offset for a dense number. If the symbol is global, the global's index is returned. If the symbol occurred in a file, the sum of all symbols in files occurring before that file and the symbol's index within the file is returned.

st_pdadd_idn
Adds an entry to the procedure table for the *st_proc entry* generated by procbegin. This should be called when the front-end generates code for the procedure in question.

The programs must be loaded with the object file access routine library libmld.a.

AUTHOR

Mark I. Himelstein

SEE ALSO

stcu(3x), stfd(3x)

NAME

stio – routines that provide a binary read/write interface to the MIPS symbol table

SYNOPSIS

```
#include <syms.h>

long st_readbinary (filename, how)
char *filename;
char how;

long st_readst (fn, how, filebase, pchdr,flags)
long fn;
char how;
long filebase;
pCHDRR pchdr;
long flags;

void st_writebinary (filename, flags)
char *filename;
long flags;

void st_writest (fn, flags)
long fn;
long flags;
```

DESCRIPTION

The CHDRR structure (see *<cmplrs/stsupport.h>* and the *stcu* manual page) represents a symbol table in memory. A new CHDRR can be created by reading a symbol table in from disk. *St_readbinary* and *st_readst* read a symbol table in from disk.

St_readbinary takes the file name of the symbol table and assumes the symbol table header *(HDRR* in *<sym.h>*) occurs at the beginning of the file. *St_readst* assumes that its file number references a file positioned at the beginning of the symbol table header and that the *filebase* parameter specifies where the object or symbol table file is based (for example, non-zero for archives).

The second parameter to the read routines can be 'r' for read only or 'a' for appending to the symbol table. Existing local symbol, line, procedure, auxiliary, optimization, and local string tables can not be appended. If they didn't exist on disk, they can be created. This restriction stems from the allocation algorithm for those symbol table sections when read in from disk and follows the standard pattern for building the symbol table.

The symbol table can be read incrementally. If *pchdr* is zero, *st_readst* assumes that no symbol table has been read yet; therefore, it reads in the symbol table header and file descriptors. The *flags* argument is a bit mask that defines what other tables should be read. *St_p** constants for each table, defined in <*cmplrs/stsupport.h*>, can be ORed. If *flags* equals '-1', all tables are read. If *pchdr* is set, the tables specified by *flags* are added to the tables that have already been read. *Pchdr*'s value can be gotten from *st_current_pchdr*. See *stcu*(3).

Line number entries are encoded on disk, the read routines expand them to longs. See the *MIPS System Programmer Guide*.

If the version stamp is out of date, a warning message is issued to stderr. If the magic number in the HDRR is incorrect, *st_error* is called. All other errors cause the read routines to read non-zero; otherwise, a zero is returned.

St_writebinary and *st_writest* are symmetric to the read routines, excluding the *how* and *pchdr* parameters. The *flags* parameter is a bit mask that defines what table should be written. *St_p* constants for each table, defined in* <*cmplrs/stsupport.h*>, can be ORed. If *flags* equals '-1', all tables are written.

The write routines write sections of the table in the approved order, as specified in the link editor (*ld*) specification.

Line numbers are compressed on disk. See the *MIPS System Programmer Guide*.

The write routines start all sections of the symbol table on four-byte boundaries.

If the write routines encounter an error, *st_error* is called. After writing the symbol table, further access to the table by other routines is undefined.

The programs must be loaded with the object file access routine library libmld.a.

AUTHOR

Mark I. Himelstein

SEE ALSO

stcu(3x), stfe(3x), stfd(3x)

NAME

stprint – routines to print the symbol table

SYNOPSIS

#include <syms.h>

#include <stdio.h>

```
char    *st_mlang_ascii [];
char    *st_mst_ascii [];
char    *st_msc_ascii [];
char    *st_mbt_ascii [];
char    *st_mtq_ascii [];
```

```
void st_dump (fd, flags)
FILE *fd;
long flags;
```

```
void st_printfd (fd, ifd, flags)
FILE *fd;
long ifd;
long flags;
```

DESCRIPTION

The *stprint* routines and arrays provide an easy way to print the MIPS symbol table. (using *st_current pchdr().*)

The arrays map constants to their ASCII equivalents. The constants can be found in *symconst.h* and represent languages (*lang*), symbol types (*st*), storage classes (*sc*), basic types (*bt*), and type qualifiers (*tq*).

The *st_dump* routine prints an ASCII version of the symbol. If fd is NULL, the routine prints file *fd* and stdout. The flags can be a mask of a section of symbol table specified by ORing *ST_P** constants together from *cmplrs/stsupport.h.* This routine modifies the current file.

St_printfd prints the sections associated with the file specified by the ifd argument. The other arguments are the same as in *st_dump.* These arguments modify the current file, as well.

AUTHOR Mark I. Himelstein

SEE ALSO

stfe(3x), stcu(3x), sym.h(5), stsupport.h(5)

BUGS

The interface will be added to incrementally as needed.

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcasecmp, strncasecmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strstr, index, rindex – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (char *s1, const char *s2);

char *strdup (const char *s1);

char *strncat (char *s1, const char *s2, size_t n);

int strcmp (const char *s1, const char *s2);

int strncmp (const char *s1, const char *s2, size_t n);

int strcasecmp (const char *s1, const char *s2);

int strncasecmp (const char *s1, const char *s2, size_t n);

char *strcpy (char *s1, const char *s2);

char *strncpy (char *s1, const char *s2, size_t n);

size_t strlen (const char *s);

char *strchr (const char *s, int c);

char *strrchr (const char *s, int c);

char *strpbrk (const char *s1, const char *s2);

size_t strspn (const char *s1, const char *s2);

size_t strcspn (const char *s1, const char *s2);

char *strtok (char *s1, const char *s2);

char *strstr (const char *s1, const char *s2);

#include <strings.h>

char *index (const char *s, int c);

char *rindex (const char *s, int c);
```

DESCRIPTION

The arguments **s1**, **s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**.

Strdup returns a pointer to a new string which is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using *malloc*(3C). If the new string can not be created, null is returned.

Strncat appends at most **n** characters. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. *Strncmp* makes the same comparison but looks at at most **n** characters. *Strcasecmp* and *strncasecmp* are identical in function, but are case-insensitive. The returned lexicographic difference reflects a conversion to lower-case.

Strcpy copies string **s2** to **s1**, stopping after the null character has been copied. The function returns **s1**.

Strncpy copies not more than **n** characters, (characters in **s2** following a null character are not copied). from the array pointed to by **s2** to the array pointed to by **s1**. If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1** until **n** characters in all have been written. The result will not be null-terminated if the length of **s2** is **n** or more. The function returns **s1**.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Index (*rindex*) are included as duplicates of *strchr* (*strrchr*) for compatibility (see Notes).

Strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

Strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way

subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

Strstr locates the first occurrence in the string **s1** of the sequence of characters (excluding the terminating null character) in the string **s2**.

All of these functions are declared in the <*string.h*> header file.

NOTES

Declarations for *index* and *rindex* are specifically omitted from <*string.h*> due to possible naming conflicts. Instead, they are declared in <*strings.h*>.

The *index*, *rindex*, *strcasecmp*, *strncasecmp* routines are from the 4.3BSD or 4.3BSD-tahoe standard C library.

SEE ALSO

malloc(3C), malloc(3X).

CAVEATS

Strcmp, *strncmp*, *strcasecmp*, and *strncasecmp* are implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high-order bit set not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

#include <stdlib.h>

double strtod (const char *str, char **ptr);

double atof (const char *str);

DESCRIPTION

strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

strtod recognizes an optional string of ''white-space'' characters [as defined by *isspace* in *ctype*(3C)], then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE (as defined in **<math.h>**) is returned (according to the sign of the value), and *errno* is set to ERANGE. If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

#include <stdlib.h>

long int strtol (const char *str, char **ptr, int base);

long int atol (const char *str);

int atoi (const char *str);

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading ''white-space'' characters [as defined by *isspace* in *ctype*(3C)] are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and ''0x'' or ''0X'' is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading ''0x'' or ''0X'' hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

CAVEAT

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*−1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

sysid – return system identifier

SYNOPSIS

unsigned sysid (id)
unsigned char id[16];

DESCRIPTION

sysid provides a unique system identifier. If *id* is non-NULL, the full 16 byte system identifier is placed in the array pointed to by *id*. This identifier is guaranteed to be unique throughout the Silicon Graphics product family.

SEE ALSO

sysinfo(1), syssgi(2).

DIAGNOSTICS

sysid returns a psuedo unique 32 bit unsigned number.

NAME

syslog, openlog, closelog, setlogmask, vsyslog – control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logopt, facility)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

setlogmask(maskpri)

#include <varargs.h>
vsyslog (priority, message, ap)
char *message;
va_list ap;

DESCRIPTION

Syslog arranges to write *message* onto the system log maintained by *syslogd*(1M). The message is tagged with *priority*. The message looks like a *printf*(3) string except that **%m** is replaced by the current error message (collected from *errno*). A trailing newline is added if needed. This message will be read by *syslogd*(1M) and written to the system console, log files, or forwarded to *syslogd* on another host as appropriate. *Vsyslog* is like *syslog* except that instead of being called with a variable number of arguments, it is called with an argument list as defined by *varargs*(5).

Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.

LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

If *syslog* cannot pass the message to *syslogd*, it will attempt to write the message on */dev/console* if the LOG_CONS option is set (see below).

If special processing is needed, *openlog* can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *Logopt* is a bit field indicating logging options.

Current values for *logopt* are:

LOG_PID	log the process id with each message: useful for identifying instantiations of daemons.
LOG_CONS	Force writing messages to the console if unable to send it to *syslogd*. This option is safe to use in daemon processes that have no controlling terminal since *syslog* will fork before opening the console.
LOG_ODELAY	Delay opening the connection to *syslogd* until the first *syslog* call. This is the default.
LOG_NDELAY	Open the connection to *syslogd* immediately. Useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Don't wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCHLD, as *syslog* may otherwise block waiting for a child whose exit status has already been collected.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as *routed*(1M), *ftpd*(1M), *rshd*(1M), etc.

LOG_AUTH — The authorization system: *login*(1), *su*(1), *getty*(1M), etc. *ftpd*(1M), and *rshd*(1M) also use LOG_AUTH.

LOG_LPR — The line printer spooling system: *lpr*(1), *lpd*(1M), etc.

LOG_LOCAL0 — Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

Closelog can be used to close the log file.

Setlogmask sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

syslogd(1M)

NAME

system – issue a shell command

SYNOPSIS

#include <stdlib.h>

int system (const char *string);

DESCRIPTION

system causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

exec(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

system forks to create a child process that in turn exec's /bin/sh in order to execute *string*. If the fork or exec fails, *system* returns a negative value and sets *errno*.

NAME

taskblock, taskunblock, tasksetblockcnt – routines to block/unblock tasks

C SYNOPSIS

```
#include <sys/types.h>
#include <task.h>

int taskblock (tid_t tid);

int taskunblock (tid_t tid);

int tasksetblockcnt (tid_t tid, int count);
```

FORTRAN SYNOPSIS

```
integer*4 function taskblock (tid)
integer*4 tid

integer*4 function taskunblock (tid)
integer*4 tid

integer*4 function tasksetblockcnt (tid, count)
integer*4 tid
integer*4 count
```

DESCRIPTION

These routines provide a complete set of blocking/unblocking capabilities for tasks. Blocking is implemented via a counting semaphore in the system. Each call to *taskblock* decrements the count and, if it goes negative, the task is suspended. When *taskunblock* is called, the count is incremented, and if it goes positive (or zero), the task is re-started. This provides both a simple, race free synchronization ability between two tasks, as well as a much more powerful capability to synchronize multiple tasks.

In order to guarantee a known starting place the *tasksetblockcnt* function may be called which will force the semaphore count to the value given by *count*. New tasks have their semaphore zeroed. Normally, *count* should be set to 0. If the resulting block count is greater than or equal to zero and the task is currently blocked, it will be unblocked. If the resulting block count is less than zero, the task will be blocked. Using this, a simple rendezvous mechanism can be set up. If one task wanted to wait for *n* other tasks to complete, it could set its block count to *-n*. This would immediately force the task to block. Then as each task finishes it unblocks the waiting task. When the *n'th* task finishes the waiting task will be woken.

A task may block another task provided that standard UNIX permissions are satisfied.

These routines will fail and no operation will be performed if one or more of the following are true:

[EINVAL]
: The *tid* specified is not a valid task id.

[EPERM]
: The caller is not operating on itself, its effective user ID is not super-user, and its real or effective user ID does not match the real or effective user ID of the to be acted on task.

SEE ALSO

blockproc(2), taskdestroy(3P), taskctl(3P), taskcreate(3P).

DIAGNOSTICS

Upon successful completion, 0 is returned. Otherwise, a value of −1 is returned to the calling task, and *errno* is set to indicate the error.

NAME

taskcreate – create a new task

C SYNOPSIS

#include <task.h>

tid_t taskcreate (char *name, void (*entry)(), void *arg,
int sched);

FORTRAN SYNOPSIS

integer *4 function taskcreate (name, entry, arg, sched)
character*(*) name
external entry
integer*4 arg
integer*4 sched

DESCRIPTION

Taskcreate causes a new task to be created for the calling process/task. The new task is created via the *sproc(2)* system call, requesting that all attributes (e.g. open files, current directory, uid, etc.) be shared.

The new task differs from the calling task as described in *sproc(2)*.

The new task will be invoked as follows:

```
entry(arg)
void *arg;
```

The *sched* parameter is currently unused and should be set to 0.

No implicit synchronization is implied between tasks - they are free to run on any processor in any sequence. Synchronization must be provided by the application using locks and semaphores (see *usinit(3P)*).

Each created task has a task block allocated. The task blocks are linked together and pointed to by *taskheader*. The task block structure is defined in *task.h*.

Taskcreate will fail and no new task will be created if one or more of the following are true:

The *sproc(2)* system call fails.

[ENOMEM] The required memory for the task block or task name was not available.

SEE ALSO

sproc(2), taskblock(3P), taskctl(3P), taskdestroy(3P).

DIAGNOSTICS

Upon successful completion, *taskcreate* returns the task id of the new task. The task id is guaranteed to be the smallest available. Otherwise, a value of –1 is returned to the calling task, and *errno* is set to indicate the error.

NAME

taskctl – operations on a task

C SYNOPSIS

#include <sys/types.h>
#include <task.h>

int taskctl (tid_t tid, unsigned option, ...);

FORTRAN SYNOPSIS

integer*4 function taskctl (tid, option)
integer*4 tid
integer*4 option

DESCRIPTION

Taskctl provides both information about tasks and the ability to control certain attributes of a task. *Option* specifies one of the following actions:

TSK_ISBLOCKED
returns 1 if the specified task is currently blocked. Since other processes could have subsequently unblocked the task, the result should be considered a snapshot only.

Taskctl will fail if one or more of the following are true:

[EINVAL] *option* does not refer to a valid option.

[EINVAL] *tid* does not refer to a valid task.

SEE ALSO

prctl(2), taskcreate(3P), taskdestroy(3P).

DIAGNOSTICS

Upon successful completion, *taskctl* returns 0 or 1. Otherwise, a value of –1 is returned to the calling task, and *errno* is set to indicate the error.

NAME

taskctl – operations on a task

C SYNOPSIS

#include <sys/types.h>
#include <task.h>

int taskctl (tid_t tid, unsigned option, ...);

FORTRAN SYNOPSIS

integer*4 function taskctl (tid, option)
integer*4 tid
integer*4 option

DESCRIPTION

Taskctl provides both information about tasks and the ability to control certain attributes of a task. *Option* specifies one of the following actions:

TSK_ISBLOCKED
returns 1 if the specified task is currently blocked. Since other processes could have subsequently unblocked the task, the result should be considered a snapshot only.

Taskctl will fail if one or more of the following are true:

[EINVAL] *option* does not refer to a valid option.

[EINVAL] *tid* does not refer to a valid task.

SEE ALSO

prctl(2), taskcreate(3P), taskdestroy(3P).

DIAGNOSTICS

Upon successful completion, *taskctl* returns 0 or 1. Otherwise, a value of −1 is returned to the calling task, and *errno* is set to indicate the error.

NAME

taskdestroy – destroy a task

C SYNOPSIS

#include <sys/types.h>
#include <task.h>

int taskdestroy (tid_t tid);

FORTRAN SYNOPSIS

integer*4 function taskdestroy (tid)
integer*4 tid

DESCRIPTION

Taskdestroy causes the named task to be destroyed. Any task within a process can destroy any other task in that process.

Taskdestroy will fail and no task will be destroyed if the following is true:

[EINVAL] The *tid* specified is not a valid task id for the calling process.

SEE ALSO

sproc(2), taskblock(3P), taskctl(3P), taskcreate(3P).

DIAGNOSTICS

Upon successful completion, *taskdestroy* returns 0. Otherwise, a value of –1 is returned to the calling task, and *errno* is set to indicate the error.

NAME

tcgetpgrp, tcsetpgrp – POSIX Get/Set Foreground Process Group Primitives

#include <sys/types.h>

int tcgetpgrp (int fildes);

int tcsetpgrp (int fildes, pid_t pgrp_id);

DESCRIPTION

The above functions retrieve and set the process group which currently owns the controlling terminal. In all cases *fildes* is the open file descriptor of a terminal file.

tcgetpgrp returns the value of the process group ID of the foreground process group associated with the terminal. This call is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

If the process has a controlling terminal, *tcsetpgrp* sets the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

DIAGNOSTICS

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned, and *errno* is set to indicate the error.

ERRORS

tcgetpgrp:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal or the file is not the controlling terminal.

tcsetpgrp:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The value of the *pgrp_id* argument is a value not supported by this implementation.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The value of *pgrp_id* is a value supported by the implementation but does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

termio(7), stty(1) in the *User's Reference Manual.*

NAME

tcsendbreak, tcdrain, tcflush, tcflow – POSIX Line Control Primitives

#include <termios.h>

int tcsendbreak (int fildes, int duration);

int tcdrain (int fildes);

int tcflush (int fildes, int queue_selector);

int tcflow (int fildes, int action);

DESCRIPTION

The above functions affect terminal line control. In all cases *fildes* is the open file descriptor of a terminal file.

tcsendbreak function causes transmission of a continuous stream of zero-valued bits for .25 seconds. In this implementation, the *duration* parameter is ignored.

tcdrain waits until all output written to the object referred to by *fildes* has been transmitted.

tcflush discards the data written but not yet transferred to the object referred to by *fildes*, or data received but not yet read, depending on the value of *queue_selector:*

1) If *queue_selector* is **TCIFLUSH**, it will flush data received but not read.
2) If *queue_selector* is **TCOFLUSH**, it will flush data written but not transmitted.
3) If *queue_selector* is **TCIOFLUSH**, it will flush both data received but not read, and data written but not transmitted.

tcflow suspends the transmission or reception of data on the object referred to by *fildes*, depending on the value of *action:*

1) If *action* is **TCOOFF**, it will suspend output.
2) If *action* is **TCOON**, it will restart suspended output.
3) If *action* is **TCIOFF**, it will transmit a STOP character, which causes the terminal device to stop transmitting data to the system.
4) If *action* is **TCION**, it will transmit a START character, which causes the terminal device to start transmitting data to the system.

DIAGNOSTICS

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

ERRORS

In all 4 function calls:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not a terminal.

tcdrain:

[EINTR]	A signal interrupted the tcdrain function.

tcflush:

[EINVAL]	The *queue_selector* argument is not a proper value.

tcflow:

[EINVAL]	The *action* argument is not a proper value.

SEE ALSO

termio(7), stty(1) in the *User's Reference Manual.*

NAME

tcsetattr, tcgetattr – POSIX Get/Set Terminal State Primitives

#include <termios.h>

int tcgetattr (int fildes, struct termios *termios_p);

int tcsetattr (int fildes, int opt_actions, struct termios *termios_p);

DESCRIPTION

tcsetattr and *tcgetattr* are used to control the general terminal functions. *Fildes* is the open file descriptor of a terminal file. *tcgetattr* fetches the parameters associated with the object referred to by *fildes* and stores them in the *termios* structure referenced by *termios_p*. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The *tcsetattr* function sets these parameters as follows:

(1) If *opt_actions* is **TCSANOW**, the change shall occur immediately.

(2) If *opt_actions* is **TCSADRAIN**, the change shall occur after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.

(3) If *opt_actions* is **TCSAFLUSH**, the change shall occur after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read shall be discarded before the change is made.

The symbolic constants for the values of *opt_actions* are defined in *<termios.h>*.

DIAGNOSTICS

Upon successful completion, a value of zero is returned. Otherwise, a value of –1 is returned, and *errno* is set to indicate the error.

ERRORS

Both functions return the following errors:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not a terminal.

tcsetattr:

[EINVAL] The *opt_actions* argument is not a proper value, or an attempt was made to change an attribute represented in the *termios* structure to an unsupported value.

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile (void);

DESCRIPTION

tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), perror(3C), stdio(3S), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (char *s);

char *tempnam (const char *dir, const char *pfx);

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

tmpnam always generates a file name using the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* [see *malloc*(3C)]. If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

CAVEATS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen to render duplication by other means unlikely.

NAME

sin, cos, tan, asin, acos, atan, atan2, fsin, fcos, ftan, fasin, facos, fatan, fatan2 – trigonometric functions and their inverses

SYNOPSIS

#include <math.h>

double sin(double x);

float fsin(float x);

double cos(double x);

float fcos(float x);

double tan(double x);

float ftan(float x);

double asin(double x);

float fasin(float x);

double acos(double x);

float facos(float x);

double atan(double x);

float fatan(float x);

double atan2(double y, double x);

float fatan2(float y, float x);

DESCRIPTION

sin, *cos* and *tan* return trigonometric functions of radian arguments x for double data types. *fsin*, *fcos* and *ftan* do the same for float data types.

asin and *fasin* return the arc sine in the range $-\pi/2$ to $\pi/2$ for double and float data types respectively.

acos and *facos* return the arc cosine in the range 0 to π for double and float data types respectively.

atan and *fatan* return the arc tangent in the range $-\pi/2$ to $\pi/2$ for double and float data types respectively.

atan2 and *fatan2* return the arctangent of *y/x* in the range $\pi-$ to π using the signs of both arguments to determine the quadrant of the return value for double and float data types respectively.

DIAGNOSTICS

If $|x| > 1$ then *asin(x)* and *acos(x)* return the default quiet NaN.

There are three math libraries. See *math*(3M) for an overview.

For -lm, atan2(0,0) returns the default quiet NaN. For -lm43 atan2(0,0) returns 0 as discussed below.

NOTES

atan2 defines *atan2(0,0)* = 0. The reasons for assigning a value to *atan2(0,0)* are these:

(1) Programs that test arguments to avoid computing *atan2(0,0)* must be indifferent to its value. Programs that require it to be invalid are vulnerable to diverse reactions to that invalidity on diverse computer systems.

(2) *atan2* is used mostly to convert from rectangular (x,y) to polar (r,θ) coordinates that must satisfy x = r*cosθ and y = r*sinθ. These equations are satisfied when (x=0,y=0) is mapped to (r=0,θ=0). In general, conversions to polar coordinates should be computed thus:

$$r := \text{hypot}(x,y); \qquad \ldots := \sqrt{(x^2+y^2)}$$
$$\theta := \text{atan2}(y,x).$$

(3) The foregoing formulas need not be altered to cope in a reasonable way with signed zeros and infinities on a machine, such as MIPS machines, that conforms to IEEE 754; the versions of hypot and atan2 provided for such a machine are designed to handle all cases. That is why *atan2*(±0,–0) = ±π, for instance. In general the formulas above are equivalent to these:

r := √(x*x+y*y); if r = 0 then x := copysign(1,x);
if x > 0 then θ := 2*atan(y/(r+x))
else θ := 2*atan((r–x)/y);

except if r is infinite then *atan2* will yield an appropriate multiple of π/4 that would otherwise have to be obtained by taking limits.

ERROR (due to Roundoff etc.)

Let P stand for the number stored in the computer in place of π = 3.14159 26535 89793 23846 26433 ... . Let "trig" stand for one of "sin", "cos" or "tan". Then the expression "trig(x)" in a program actually produces an approximation to trig(x*π/P), and "atrig(x)" approximates (P/π)*atrig(x). The approximations are close.

In the codes that run on MIPS machines, P differs from by a fraction of an **ulp**; the difference matters only if the argument x is huge, and even then the difference is likely to be swamped by the uncertainty in x. Besides, every trigonometric identity that does not involve π explicitly is satisfied equally well regardless of whether P = π. For instance, $\sin^2(x)+\cos^2(x) = 1$ and *sin*(2x) = 2 sin(x)cos(x) to within a few **ulps** no matter how big x may be. Therefore the difference between P and π is most unlikely to affect scientific and engineering computations.

SEE ALSO

math(3M), hypot(3M), sqrt(3M)

AUTHOR

Robert P. Corbett, W. Kahan, Stuart I. McDonald, Peter Tang and, for the codes for IEEE 754, Dr. Kwok–Choi Ng.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

void *tsearch (const void *key, void **rootp,
int (*compar)(const void *, const void *));

void *tfind (const void *key, void **rootp,
int (*compar)(const void *, const void *));

void *tdelete (const void *key, void **rootp,
int (*compar)(const void *, const void *));

void twalk ((void *root, void (*action)());

DESCRIPTION

tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

Tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is

the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

struct node {           /* pointers to these are stored in the tree
        char *string;
        int length;
};
char string_space[10000]; /* space to store strings */
struct node nodes[500];   /* nodes to store */
struct node *root = NULL;         /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, (char **) &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
```

```
                nodeptr++;
        }
        twalk((char *)root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
        return strcmp(((struct node *)node1)->string,
        ((struct node *) node2)->string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/
void
print_node(node, order, level)
char **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf)  {
                (void)printf("string = %20s,  length = %d\n",
                    (*((struct node **)node))->string,
                    (*((struct node **)node))->length);
        }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tfind* and *tdelete* if **rootp** is NULL on entry.
If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

#include <unistd.h>

char *ttyname (int fildes);

int isatty (int fildes);

DESCRIPTION

ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by calling *ttyname*(3C) to determine which device the calling program has associated with the standard input, the standard output, or the error output (0, 1 or 2). This device name is then searched for in the **/etc/utmp** file.

FILES

/dev
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of -1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

handle_unaligned_traps, print_unaligned_summary – gather statistics on unaligned references

SYNOPSIS

```
void handle_unaligned_traps()

void print_unaligned_summary()

long unaligned_load_word(addr)
char *addr;

long unaligned_load_half(addr)
char *addr;

long unaligned_load_uhalf(addr)
char *addr;

float unaligned_load_float(addr)
char *addr;

double unaligned_load_double(addr)
char *addr;

void unaligned_store_word(addr, value)
char *addr;
long value;

void unaligned_store_half(addr, value)
char *addr;
long value;

void unaligned_store_float(addr, float value)
char *addr;
float value;

void unaligned_store_double(addr, value)
char *addr;
double value;
```

DESCRIPTION

The first two routines implement a facility for finding unaligned references. The MIPS hardware traps load and store operations where the address is not a multiple of the number of bytes loaded or stored. Usually this trap indicates incorrect program operation and so by default the kernel converts this trap into a SIGBUS signal to the process, typically causing a core dump for debugging.

Older programs developed on systems with lax alignment constraints sometimes make occasional misaligned references in course of correct operation. The best way to port such programs to MIPS hardware is to correct the program by aligning the data.

A call to *handle_unaligned_traps* installs a SIGBUS handler that fixes unaligned memory references and keeps a record of the types, counts, and instruction addresses of these traps. A call to *print_unaligned_summary* prints the accumulated information. The following is an example of the output produced by *print_unaligned_summary*:

```
#################################
#       unaligned reference summary #
# byte aligned lw        5000  33.3%  #
# byte aligned sw       10000  66.7% #
# 0x0040024c/i           5000  33.3%  33.3%#
# 0x004002a8/i           5000  33.3%  66.7%#
# 0x004002b4/i           5000  33.3% 100.0%#
#################################
```

The listing is written to standard error and describes the type and number of unaligned references, followed by a list of every address that contains an unaligned reference. To convert the addresses into a *dbx*(1) script and run the script, pipe the output (both standard output and standard error) through the following command. The output from **dbx** will be the name of the function and line number of the misalignment.

```
sed -n -e 's;^ # [0-9a-f]*/i).*#$;1;p' | dbx prog
```

This information can be used to decide the best way to correct the problem. If not all of the data can be aligned, or not all of the identified program locations that reference unaligned data can be changed, the *sysmips*(2) [MIPS_FIXADE] system call may be appropriate.

The other routines load or store their indicated data type at the address specified. The address need not meet the normal alignment constraints.

There exist FORTRAN entry points for these routines so they may be called directly from FORTRAN with the names documented here.

Programs using these functions must be loaded with */usr/lib/fixade.o*.

DIAGNOSTICS

If these routines try to load or store to an address that is outside the program's address space a SIGSEGV signal will be generated from inside these routines. If the program did not use these routines and the address was unaligned then the program would generate a SIGBUS signal. This is because the check for alignment is done before the address is checked to be in the program's address space.

FILES

/usr/lib/fixade.o

SEE ALSO

dbx(1), sysmips(2) [MIPS_FIXADE], signal(2), sigset(2).

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (int c, FILE *stream);

DESCRIPTION

ungetc inserts the character *c* into the buffer associated with an input *stream*. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (*fseek*,or *rewind*), discards any pushed-back characters. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals **EOF**, the operation fails and the input stream is unchanged. A successful call to *ungetc* clears the end-of-file indicator for the stream.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

ungetc returns the character pushed back, or **EOF** if it cannot insert the character.

BUGS

When *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

NAME

unregisterhost – remove the existing host entry in yp hosts data base

SYNOPSIS

int unregisterhost(name, passwd)
char *name, *passwd;

DESCRIPTION

Unregisterhost sends an host name unregister request to *registrar(1M)* on yp master via the *yp_update(3R)* call. After successfully completed, the host name will no longer be in the yp hosts data base and its internet address is freed. This routine should be used only when yellow page service is enabled in the network.

The arguments for the routine are:

name The host name of the entry to be deleted.

passwd The root password of yp master. If yp master does not have root password, simply pass a NULL.

Unregisterhost returns NULL when successfully completed. All error codes are defined in <rpcsvc/ypclnt.h>.

Unregisterhost always wait until yellow page data base are pushed to all slave servers.

SEE ALSO

registrar(1M), yp_update(1M), registerinethost(3N), renamehost(3N), yppush(1M)

AUTHOR

Steve Sun

NAME

usconfig – semaphore and lock arena configuration operations

C SYNOPSIS

#include <ulocks.h>

int usconfig (int cmd [, int arg0 [, int arg1]]);

FORTRAN SYNOPSIS

#include <ulocks.h>

integer*4 function usconfig (cmd [, arg0 [, arg1]])
integer*4 cmd, arg0 integer*4 arg1(8)

DESCRIPTION

usconfig is used to configure the use of semaphores and locks. Some of these options set configurable parameters to be used on the next *usinit*(3P), others give back information about a particular arena. The following *cmds* are available:

CONF_INITSIZE Sets the shared segment size (in bytes) for semaphores, locks, and the usmalloc arena to the value given by *arg0*. The default is 65536 bytes. This only has effect if called before a *usinit*(3P). It returns the previously set value.

CONF_INITUSERS Sets the maximum number of users for a given group of semaphores and locks to the value given by *arg0*. The maximum allowable is 512 users, and the default is 8. This only has effect if called before *usinit*(3P). Each process that calls *usinit*(3P) is considered a user, as is each process that uses a spinlock. It returns the previously set value.

CONF_GETSIZE Returns the arena size (in bytes) for the arena specified by *arg0* (as returned by *usinit*(3P)).

CONF_GETUSERS Returns the maximum number of users for the arena specified by *arg0* (as returned by *usinit*(3P)).

CONF_LOCKTYPE *arg0* defines which of **US_NODEBUG**, **US_DEBUG**, or **US_DEBUGPLUS** locks are to be used in the arena set up by the next call to *usinit*(3P). The **US_NODEBUG** option is the fastest, and no debugging or metering information is available. **US_DEBUG** locks provide information about each lock transaction. The metering information gathered consists of a count of the number of times the lock is requested and found locked, ''spins'' and the number of times the lock is acquired, ''hits''. All

metering is stored in a *lockmeter_t* structure and retrievable via *usctllock*(3P)). The debugging information maintained consists of the process id, "pid" of the owner of the lock. The pid is set to -1 if no one owns the lock. All debug info is stored in a *lockdebug_t* structure and retrievable via *usctllock*(3P)). The **US_DEBUGPLUS** option provides the same debugging and metering information, except that in addition, if either a unset lock is unlocked, a set lock is unlocked by other than the setter, or a lock is locked twice by the same caller, a message is printed to *stderr*.

CONF_ARENATYPE By default, arenas are configured so that unrelated processes may join the arena by specifying the appropriate file name when calling *usinit*(3P). This means that the file must continue to exist for the duration of the time the arena is in use. If the file is a temporary file, it may be difficult for an application program to guarantee the file gets removed at the appropriate time. By specifying the arena type (via *arg0*) to be **US_SHAREDONLY** then *usinit*(3P) will unlink the file after it has opened it. This of course means that unrelated processes may NEVER join the arena. Setting an arena to be **US_SHAREDONLY** has an additional effect for the software spinlock implementation - only one open file descriptor will be maintained for the entire collection of shared processes accessing any given arena. Normally, each process (shared or unrelated) will keep an open file descriptor to the arena.

CONF_HISTON Enable semaphore history logging for the arena given by *arg0*. The history mechanism may then be enabled for previously allocated semaphores using *usctlsema*(3P). All subsequent semaphores allocated via *usnewsema*(3P) are set to log their history. This *cmd* serves as a global flag on the history mechanism in conjunction with **CONF_HISTOFF** to allow for quick enabling and disabling of history. The history mechanism logs the operation, the semaphore for which the operation was done, the pid of the process performing the operation, and the address from which the operation was called. No history is maintained for locks, since the number of transactions on

locks is typically large.

CONF_HISTFETCH Fills a ''histptr_t'' structure pointed to by *arg1* (defined in *<ulocks.h>*). This structure contains the number of entries in the history list, and a pointer to the most recent history structure. The history list is a doubly linked list, so that the user can then traverse the list as they see fit. The ''hist_t'' structure (also defined in *<ulocks.h>*) is described below. *arg0* is the arena pointer as returned by *usinit*(3P).

CONF_HISTOFF Disable the history mechanism for all semaphores in the arena defined by *arg0*. Note that this simply turns off a global history flag for the given arena - the invidual semaphores' history state is unaffected.

CONF_HISTRESET Reinitializes the hist_t chain for the arena given in *arg0* to contain no entries. This frees all previously allocated history records.

CONF_STHREADIOOFF
By default, the *stdio*(3) routines available with *libmpc.a* are single threaded. Multiple shared address space processes may attempt to execute them simultaneously and the system guarantees that they will work as expected. This requires that the *stdio*(3) data structures be locked on each access, thereby adding overhead which may be unnecessary in certain applications. This command turns off any single threading of the following routines: *getc, putc, fgetc, fputc, ungetc, getw, putw, gets, fgets, puts, fputs, fopen, fdopen, freopen, ftell, rewind, setbuf, setvbuf, fclose, fflush, fread, fwrite, fseek, popen, pclose, dup2, printf, fprintf, vprintf, vfprintf, scanf, fscanf.* The previous state of *stdio*(3) single threading is returned.

CONF_STHREADIOON
This option enables single threading of the *stdio*(3) routines. The previous state of *stdio*(3) single threading is returned.

CONF_STHREADMISCOFF
Some routines besides *stdio*(3) routines are also single threaded by default. This option disables this for the following routines: *opendir, readdir, scandir, seekdir, closedir, telldir, srand, rand.* The previous

state of single threading is returned.

CONF_STHREADMISCON

This option enables single threading of the miscellaneous routines mentioned above. This command is the inverse of **CONF_STHREADMISCOFF**. The previous state of single threading of the miscellaneous routines is returned.

CONF_STHREADMALLOCCOFF

The *malloc*(3) routines are single threaded by default. This option disables single threading for the following routines: *malloc, free, realloc, calloc, mallopt, mallinfo*. The previous state of their single threading is returned.

CONF_STHREADMALLOCCON

This option enables single threading of the *malloc*(3) routines. The previous state of single threading is returned.

Declarations of the function, *cmds*, and the **hist_t** structure, are in the *<ulocks.h>* header file.

The **CONF_INITSIZE**, **CONF_ARENATYPE**, **CONF_LOCKTYPE**, and **CONF_INITUSERS** only take effect if the caller is the process that first sets up the arena. If the process is just joining an existing arena, the settings of these parameters is ignored.

The structure declaration of **hist_t** is:

```
typedef struct                    {
        struct usema_s *h_sem;    /* the semaphore */
        int h_op;                 /* the operation */
        int h_pid;                /* the thread process id */
        int h_scnt;               /* the value of the semaphore */
        int h_wpid;               /* the waking process id */
        char *h_cpc;              /* the calling PC */
        struct hist_s *h_next;    /* the next hist_t in the chain */
        struct hist_s *h_last;    /* the previous hist_t in the chain */
} hist_t;
```

The structure declaration of **histptr_t** is:

```
typedef struct histptr_s          {
        hist_t *hp_current;       /* pointer to the last hist_t */
        int hp_entries;           /* count of hist_t structs */
        int hp_errors;            /* # of errors due to lack of space */
} histptr_t;
```

usconfig will fail if one or more of the following are true:

[EINVAL]	*cmd* is not a valid command.
[EINVAL]	*cmd* is equal to **CONF_INITSIZE** and *arg0* is less than the system-imposed minimum (4096 bytes) or greater than the system-imposed maximum size for a mapped memory segment.
[EINVAL]	*cmd* is equal to **CONF_INITUSERS** and *arg0* is greater than the system-imposed maximum (512).
[EINVAL]	*cmd* is equal to **CONF_ARENATYPE** and *arg0* is not equal to either **US_SHAREDONLY** or **US_GENERAL.**
[EINVAL]	*cmd* is equal to **CONF_HISTFETCH** and history is not currently enabled.

SEE ALSO

usinit(3P), usctlsema(3P), usinitsema(3P), usnewsema(3P), usinitlock(3P), usnewlock(3P), usctllock(3P), usmalloc(3P).

DIAGNOSTICS

Upon successful completion, the return value is dependent on the particular command. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

uscpsema – attempts to acquire a semaphore, and fails if not possible

C SYNOPSIS

#include <ulocks.h>

int uscpsema (usema_t sema);

FORTRAN SYNOPSIS

integer*4 function uscpsema (sema)
integer*4 sema

DESCRIPTION

uscpsema attempts to acquire a previously allocated semaphore specified by *sema.* If the semaphore is not available (its count is less than zero), *uscpsema* returns.

SEE ALSO

usinitsema(3P), usnewsema(3P), usvsema(3P), uspsema(3P), ustestsema(3P).

DIAGNOSTICS

uscpsema returns a 0 if the semaphore is not acquired and 1 if the semaphore is acquired.

NAME

usctllock – lock control operations

C SYNOPSIS

#include <ulocks.h>

int usctllock (ulock_t lock, int cmd [, void *arg]);

FORTRAN SYNOPSIS

#include <ulocks.h>

integer*4 function usctllock (lock, cmd [, arg])
integer*4 lock
integer*4 cmd
integer*4 arg(3)

DESCRIPTION

usctllock provides a variety of lock control operations as specified by *cmd*. Metering and debugging information is available only for locks allocated out of an arena which has as a lock type either US_DEBUG or US_DEBUGPLUS (see *usconfig(3P)*). The following *cmds* are available:

CL_METERFETCH Fills the structure pointed to by *arg* with the metering data associated with *lock*.

CL_METERRESET Reinitializes the *lockmeter_t* structure associated with *lock* to all values of -1.

CL_DEBUGFETCH Fills the structure pointed to by *arg* with the debugging data associated with *lock*.

CL_DEBUGRESET Reinitializes the elements of the *lockdebug_t* structure associated with *lock* to values of -1.

Declarations of the function and *cmds*, the **lockmeter_t** structure, and the **lockdebug_t** structure, are in the *<ulocks.h>* header file.

The structure declaration of **lockmeter_t** is:

```
typedef struct          {
        int lm_spins;   /* number of times the lock was spun on */
        int lm_tries;   /* number of times the lock was requested */
        int lm_hits;    /* number of times the lock was acquired */
} lockmeter_t;
```

The structure declaration of **lockdebug_t** is:

```
typedef struct          {
        int ld_owner_pid;  /* the process that owns the lock */
} lockdebug_t;
```

An invalid *lock* may yield unpredictable results.

usctllock will fail if one or more of the following are true:

[EINVAL] *cmd* is not a valid command.

[EINVAL] *cmd* is equal to **CL_METERFETCH** and metering is not currently enabled for the given lock.

[EINVAL] *cmd* is equal to **CL_DEBUGFETCH** and debugging is not currently enabled for the given lock.

SEE ALSO

usconfig(3P), usinitlock(3P), usnewlock(3P).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

usctlsema – semaphore control operations

C SYNOPSIS

#include <ulocks.h>

int usctlsema (usema_t *sema, int cmd [, void *arg]);

FORTRAN SYNOPSIS

#include <ulocks.h>

integer*4 function usctlsema (sema, cmd [, arg])
integer*4 sema
integer*4 cmd
integer*4 arg(6)

DESCRIPTION

usctlsema provides a variety of semaphore control operations as specified by *cmd*. The following *cmds* are available:

CS_METERON Enable metering for the semaphore specified by *sema*. The metering information gathered consists of the number of *uspsema* and *uscpsema* calls, the number of times the semaphore could immediately be acquired, the number of *usvsema* calls, the number of times *usvsema* was called and no process was queued waiting, the current number of processes waiting on the semaphore, and the maximum number of processes ever waiting on the semaphore. All metering is stored in a *semameter_t* structure defined in the header file *<ulocks.h>* and described below.

CS_METEROFF Disable metering for the semaphore specified by *sema*.

CS_METERFETCH Fills the structure pointed to by *arg* with the metering data associated with *sema*.

CS_METERRESET Reinitializes the *semameter_t* structure associated with *sema* to all 0.

CS_DEBUGON Enable debug monitoring for the semaphore specified by *sema*. The debugging information maintained consists of the process id of the owner of the semaphore, and the address in the owner process where the call to the semaphore operation was made, the process id of the last process to operate on the semaphore, and the address in the last process where the call to the semaphore operation was

made. The pid is set to -1 if no one owns the semaphore. All debug info is stored in a *semadebug_t* structure defined in the header file <*ulocks.h*> and described below.

CS_DEBUGOFF Disable debugging for the semaphore specified by *sema*.

CS_DEBUGFETCH Fills the structure pointed to by *arg* with the debugging data associated with *sema*.

CS_DEBUGRESET Reinitializes the elements of the *semadebug_t* structure associated with *sema* to values of -1.

CS_HISTON Enable history logging for the semaphore specified by *sema*. A global history is maintained that consists of a record of each transaction on semaphores in structures defined in the header file <*ulocks.h*>. This is discussed further in *usconfig*(3P), which is used to retrieve history of semaphore transactions.

CS_HISTOFF Disable history for the semaphore specified by *sema*.

Declarations of the function and *cmds*, the **hist_t** structure, the **semameter_t** structure, and the **semadebug_t** structure, are in the <*ulocks.h*> header file.

The structure declaration of **semameter_t** is:

```
typedef struct semameter_s {
        int sm_phits;        /* number of successful psemas*/
        int sm_psemas;       /* number of psema attempts */
        int sm_vsemas;       /* number of vsema attempts */
        int sm_vnowait;      /* number of vsemas with no one
                                waiting */
        int sm_nwait;        /* number of threads waiting on the
                                semaphore */
        int sm_maxnwait;     /* maximum number of threads waiting
                                on the semaphore */
} semameter_t;
```

The structure declaration of **semadebug_t** is:

```
typedef struct semadebug_s {
        int sd_owner_pid     /* the process that owns the semaphore */
        char * sd_owner_pc;  /* the address of last psema for
                                  process that owns the semaphore */
        int sd_last_pid;     /* the process that last did a p or v sema */
        char * sd_last_pc;   /* the address of the last routine to operate
```

```
                                      on the semaphore */
    } semadebug_t;
```

usctlsema will fail if one or more of the following are true:

[EINVAL] *cmd* is not a valid command.

[EINVAL] *cmd* is equal to **CS_METERFETCH** and metering in not currently enabled.

[EINVAL] *cmd* is equal to **CS_DEBUGFETCH** and debugging is not currently enabled.

[ENOMEM] *cmd* is equal to **CS_METERON** or **CS_DEBUGON** and there was not enough memory in the arena.

SEE ALSO

usconfig(3P), usinitsema(3P), usnewsema(3P), uscpsema(3P), uspsema(3P), usvsema(3P), usdumpsema(3P).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

usdumplock – dump out information about a specific lock

C SYNOPSIS

#include <ulocks.h>
#include <stdio.h>

int usdumplock (ulock_t lock, FILE *fd, FILE *str);

FORTRAN SYNOPSIS

integer*4 function usdumplock (lock, fd, str)
integer*4 lock, fd
character *(*) str

DESCRIPTION

usdumplock dumps information about *lock* in a readable form. This information is written to the file descriptor given by *fd*. The information printed includes where in memory the lock resides, whether it is locked or free, and the metering and debugging information (see *usctllock*). The argument *str* is simply printed as a string, and can be used to aid in identifying where *usdumplock* was called from.

SEE ALSO

usinit(3P), usctllock(3P).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

usdumpsema – dump out information about a specific semaphore

C SYNOPSIS

#include <ulocks.h>
#include <stdio.h>

int usdumpsema (usema_t sema, FILE *fd, FILE *str);

FORTRAN SYNOPSIS

integer*4 function usdumpsema (sema, fd, str)
integer*4 sema, fd
character *(*) str

DESCRIPTION

usdumpsema dumps information about *sema* in a readable form. This information is written to the file descriptor given by *fd*. The information printed includes where in memory the semaphore resides, what its count is, and the metering and debugging information (see *usctlsema*). The argument *str* is simply printed as a string, and can be used to aid in identifying where *usdumpsema* was called from.

SEE ALSO

usinit(3P), usctlsema(3P).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

usfreelock – free a lock

C SYNOPSIS

#include <ulocks.h>

void usfreelock (ulock_t lock, usptr_t *handle);

FORTRAN SYNOPSIS

subroutine usfreelock (lock, handle)
integer*4 lock
integer*4 handle

DESCRIPTION

usfreelock frees all memory associated with the lock specified by *lock*.

usfreelock will cause unpredictable results if *lock* is not a valid lock.

SEE ALSO

usctllock(3P), ussetlock(3P), usconfig(3P), usinitlock(3P), usnewlock(3P).

DIAGNOSTICS

usfreelock returns no value.

NAME

usfreesema – free a semaphore

C SYNOPSIS

#include <ulocks.h>

void usfreesema (usema_t *sema, usptr_t *handle);

FORTRAN SYNOPSIS

subroutine usfreesema (sema, handle)
integer*4 sema
integer*4 handle

DESCRIPTION

usfreesema frees a previously allocated semaphore specified by *sema*.

SEE ALSO

usinitsema(3P), usnewsema(3P), usinit(3P).

DIAGNOSTICS

usfreesema returns no value.

NAME

usgetinfo, usputinfo – exchange information though an arena

C SYNOPSIS

#include <ulocks.h>

void usputinfo (usptr_t *handle, void *info);

void *usgetinfo (usptr_t *handle);

FORTRAN SYNOPSIS

subroutine usputinfo (handle, info)
integer*4 handle, info

integer*4 usgetinfo (handle)
integer*4 handle

DESCRIPTION

When unrelated processes decide to share an arena, it is often useful to be able to initially communicate the location of various data structures within the arena. A single word communication area is available inside the arena header block, accessible via the funcitons *usgetinfo* and *usputinfo*. Thus, a process that sets up the arena can initialize various locks, semaphores, and common data structures, and place a single pointer that any process that joins the arena can retrieve. *usputinfo* places the data item in the header, overwriting any existing information there. *usgetinfo* will retrieve that information.

SEE ALSO

usinit(3P).

NAME

usinit, _utrace – semaphore and lock initialization routine

C SYNOPSIS

#include <ulocks.h>

usptr_t *usinit (char *filename);

extern int _utrace;

FORTRAN SYNOPSIS

integer*4 function usinit (filename)
character*(*) filename

subroutine ussettrace ()

subroutine usclrtrace ()

DESCRIPTION

usinit is used to initialize a shared arena from which related or unrelated processes may share semaphores, locks and memory. It must be called before any locks or semaphores can be allocated. Locks, semaphores and memory can then be allocated using the *usptr_t* returned by *usinit*. More than one call can be made to *usinit* to create separate *arenas* of locks and semaphores. In fact, calls to *usinit* may be made on behalf of a process: when *sproc*(2) is called, an arena containing the locks and semaphores for libc is created; when *m_fork*(3P) or *taskcreate*(3P) is called, an arena is set up to control the spawned tasks. *usinit* expects a filename as an argument to represent the *key* to the arena, so that it can be used between unrelated processes.

usinit creates a file, *filename*, and grows it to be a specific size. *usconfig*(3P) may be used to set or get this size. The overall size will limit how many locks and semaphores may be allocated. In addition to the basic lock and semaphore data structures, all history, metering and debug structures are also allocated via *usmalloc*(3P) from this area. File locks (see *fcntl*(2)) are used to prevent conflicting accesses to this area during the *usinit* call.

Unrelated processes can share arenas by each issuing a *usinit* call with the same *filename*. At that point, any locks, semaphores and memory in the arena may be shared.

Related processes (those created by *sproc*(2)) are automatically made 'members' of the arena and may immediately use any previously allocated locks, semaphores or memory.

Certain attributes of the newly created arena may be set prior to the call *usinit* by *usconfig*(3P).

When called, *usinit* attempts to determine whether the arena is active (i.e. whether any other processes are currently using it). This determination is made by checking whether any file locks are currently active on the file. If so, the caller registers its file lock and merely 'joins' the collection of processes using that arena. If there are no file locks, the caller re-initializes the entire arena. Problems can result if a process that did not call *usinit* is still accessing the arena (namely a child of a *sproc* whose parent has died) when a new process attempts to join. The new process will find no file locks and re-initialize the arena, thus destroying any state the first process had. A process can force registration by calling *usinit*.

usinit and the other lock and semaphore routines normally perform their functions in silence. For a verbose 'trace' of what is being allocated, the global flag **_utrace** may be set to non-zero. This may aid in debugging the various error returns.

usinit may fail for a variety of reasons. Some are dependent on the number of arenas a process has and the order in which they are initialized. In particular, all processes sharing an arena must be able to attach them at the same virtual address. If *usinit* fails, it is a good idea to set the tracing variable **_utrace** to 1. This will provide more descriptive error messages.

usinit will fail if one or more of the following are true:

	The *filename* argument could not be opened or created for read/write.
[ENOSPC]	The file specified by *filename* could not be grown to the specified size.
[ENOMEM]	There is not enough space in the arena to allocate the initial set of required locks and semaphores. The size of the arena may be manipulated with *usconfig(3P)*.
[EBUSY]	The caller already has one or more arenas initialized, and the arena to be joined requires a different lock type.
[EBUSY]	The caller already has mapped virtual space at the address required by the arena when attempting to join the arena.
	Errors may also be the result of a *mmap*(2) or a *fcntl*(2) system call.

SEE ALSO

fcntl(2), mmap(2), sproc(2), oserror(3C), usmalloc(3P), usconfig(3P), usgetinfo(3P).

DIAGNOSTICS

Upon successful completion, a pointer to a *usptr_t* structure is returned. Otherwise, a value of NULL is returned and *errno* is set to indicate the error.

NAME

usinitlock – initializes a lock

C SYNOPSIS

#include <ulocks.h>

int usinitlock (ulock_t lock);

FORTRAN SYNOPSIS

integer*4 function usinitlock (lock)
integer*4 lock

DESCRIPTION

usinitlock initializes the lock specified by *lock*. All previous information associated with the lock (metering, debugging) is reinitialized. *usinitlock* should only be used for previously allocated locks. Locks are allocated using *usnewlock(3P).*

usinitlock will cause unpredictable results if *lock* does not point to a a valid lock.

NOTE

usinitlock assumes that the fields of the ulock_t structure are in a valid state. The use of *malloc* rather than *usnewlock* may result in a segmentation violation.

SEE ALSO

usnewlock(3P), usctllock(3P).

DIAGNOSTICS

0 is always returned.

NAME

usinitsema – initializes a semaphore

C SYNOPSIS

#include <ulocks.h>

int usinitsema (usema_t *sema, int val);

FORTRAN SYNOPSIS

integer*4 function usinitsema (sema, val)
integer*4 sema, val

DESCRIPTION

usinitsema initializes the semaphore specified by *sema* to the value specified by *val.* Metering and debugging are reinitialized and the history logging mechanism is set according to the global setting (see *usconfig*(3P)). *usinitsema* should only be used for semaphores previously allocated using *usnewsema*(3P). Note that *usinitsema* does not check whether any process is currently waiting for the semaphore. Any such information is lost.

SEE ALSO

usnewsema(3P), usctlsema(3P), usfreesema(3P).

DIAGNOSTICS

0 is always returned.

NAME

usmalloc, usfree, usrealloc, uscalloc, usmallopt, usmallinfo – user shared memory allocator

C SYNOPSIS

```
#include <ulocks.h>
#include <malloc.h>

void *usmalloc (size_t size, usptr_t *handle);

void usfree (void *ptr, usptr_t *handle);

void *usrealloc (void *ptr, size_t size, usptr_t *handle);

void *uscalloc (size_t nelem, size_t elsize, usptr_t *handle);

int usmallopt (int cmd, int value, usptr_t *handle);

struct mallinfo usmallinfo (usptr_t *handle);
```

FORTRAN SYNOPSIS

```
#include <ulocks.h>

integer*4 function usmalloc (size, handle)
integer*4 size, handle

subroutine usfree (ptr, handle)
integer*4 ptr, handle

integer*4 function usrealloc (ptr, size, handle)
integer*4 ptr, size, handle

integer*4 function uscalloc (nelem, elsize, handle)
integer*4 nelem, elsize, handle

integer*4 function usmallopt (cmd, value, handle)
integer*4 cmd, value, handle
```

DESCRIPTION

These routines provide a simple general-purpose memory allocation package that allows the user to allocate from a shared arena. The shared arena is initialized using *usinit*(3P) as follows:

```
usconfig(CONF_INITSIZE, sizeYouWantTheArenaToBe);

sharedhandle = usinit("SomeLocalFile");
```

More than one call can be made to *usinit*(3P) to set up separate malloc arenas. The file name passed to *usinit*(3P) is used as a key to allow shared arenas to be created for use amongst unrelated processes. Once the arena is set up, calls to *usmalloc* will attempt to allocate space from the arena. If the arena gets full, **NULL** is returned. Note that this malloc arena is also used

by other *us** calls (such as *usnewlock* and *usnewsema*).

The argument to *usfree* is a pointer to a block previously allocated by *usmalloc*; after *usfree* is performed this space is made available for further allocation.

Undefined results will occur if the space assigned by *usmalloc* is overrun or if some random number is handed to *usfree*.

usrealloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *usrealloc* will ask *usmalloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

uscalloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

usmallopt provides for control over the allocation algorithm. See *amalloc*(3P) for details on the allowable options.

usmallinfo provides instrumentation describing space usage. See *amalloc*(3P) for details on the returned information.

SEE ALSO

usinit(3P), usconfig(3P), amalloc(3P), malloc(3X).

DIAGNOSTICS

usmalloc, *uscalloc*, and *usrealloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. If *usmallopt* is called after any allocation (for most *cmd* arguments) or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

NAME

usnewlock – allocates and initializes a lock

C SYNOPSIS

#include <ulocks.h>

ulock_t usnewlock (usptr_t *handle);

FORTRAN SYNOPSIS

integer*4 function usnewlock (handle)
integer*4 handle

DESCRIPTION

usnewlock allocates a lock from the arena designated by *handle* (returned from *usinit*(3P)) and initializes it and all associated data. There are different types of locks; by default the fastest lock type for the class of machine the process is running on is allocated. See *usconfig*(3P) for other specifiable lock types. Metering and debugging are only enabled if the locks are of the debugging type (see *usconfig*(3P). There is a limit of a maximum of 4096 locks per shared area (for hardware supported locks) and a limit based on the size of the shared arena for software locks.

usnewlock will fail if one of the following is true:

[ENOMEM] There is no memory available to allocate the lock structure.

[ENOSPC] If the maximum number of allocatable locks has been exceeded.

[ENOSPC] All the locks in the system have been allocated.

SEE ALSO

usinit(3P), usconfig(3P), usinitlock(3P), usctllock(3P).

DIAGNOSTICS

Upon successful completion, a ulock_t structure is returned, otherwise a NULL is returned and *errno* is set to indicate the error.

NAME

usnewsema – allocates and initializes a semaphore

C SYNOPSIS

#include <ulocks.h>

usema_t *usnewsema (usptr_t *handle, int val);

FORTRAN SYNOPSIS

integer*4 function usnewsema (handle, val)
integer*4 handle
integer*4 val

DESCRIPTION

usnewsema allocates a semaphore and initializes it to the value specified by *val*. Initially, metering and debugging are off (and can be turned on through a call to *usctlsema*(3P)) and the history logging mechanism is set according to the global setting (see *usconfig*(3P)). The semaphore is allocated from the shared arena designated by *handle* as returned from *usinit*(3P).

usnewsema will fail if the following is true:

[ENOMEM] There is no memory available to allocate the semaphore structure.

SEE ALSO

amalloc(3P), usinit(3P), usconfig(3P), usctlsema(3P), usfreesema(3P).

DIAGNOSTICS

Upon successful completion, a value of pointer to a *usema_t* structure is returned. Otherwise, a value of NULL is returned and *errno* is set to indicate the error.

NAME

uspsema – attempt to acquire a semaphore

C SYNOPSIS

#include <ulocks.h>

int uspsema (usema_t *sema);

FORTRAN SYNOPSIS

integer*4 function uspsema (sema)
integer*4 sema

DESCRIPTION

uspsema decrements the value of a previously allocated semaphore specified by *sema*. If the value is negative, the semaphore will block the calling process until the value goes non-negative due to a *usvsema*(3P) call made by another process. *uspsema* uses *blockproc*(2) to perform the actual suspending of the caller if necessary.

uspsema will fail if one or more of the following are true:

SEE ALSO

blockproc(2), usinitsema(3P), usnewsema(3P), usvsema(3P), uscpsema(3P).

DIAGNOSTICS

Upon successful completion the semaphore has been acquired and a value of 1 is returned.

NAME

ussetlock, uscsetlock, uswsetlock, ustestlock, usunsetlock – spinlock routines

C SYNOPSIS

#include <ulocks.h>

int ussetlock (ulock_t lock);

int uscsetlock (ulock_t lock, unsigned spins);

int uswsetlock (ulock_t lock, unsigned spins);

int ustestlock (ulock_t lock);

int usunsetlock (ulock_t lock);

FORTRAN SYNOPSIS

integer*4 function ussetlock (lock)
integer*4 lock

integer*4 function uscsetlock (lock, spins)
integer*4 lock
integer*4 spins

integer*4 function uswsetlock (lock, spins)
integer*4 lock
integer*4 spins

integer*4 function ustestlock (lock)
integer*4 lock

integer*4 function usunsetlock (lock)
integer*4 lock

DESCRIPTION

This set of routines provide a standard test and set facility. If the lock is already locked, the caller will optionally spin attempting to acquire the lock. After a configurable number of attempts, control of the processor will be relinquished, thus allowing other processes to run. At some future time, the caller will again be run, and again attempt to acquire the lock. Note that if a process spends much of its time waiting for a lock without giving up the processor, then the total throughput of the system may be reduced. On the other hand, by giving up the processor too quickly, there is a longer latency between when the lock is freed and the caller obtains the lock.

The actual algorithm used to implement these functions depends on the underlying hardware. Certain **4D** systems have hardware locks that are accessible from user processes. Others have no hardware and so the locks are implemented using Dijkstra's software lock algorithm. The choice of which kind of locks to use is made at run-time. In addition to the

underlying lock primitive, different levels of debugging information can be requested via *usconfig*(3P).

ussetlock spins until the lock specified by *lock* is acquired. *uscsetlock* conditionally attempts to acquire a lock, returning a 1 if the lock was acquired, and a 0 if it was not free. *uswsetlock* is similar to *ussetlock,* except that the user specifies the number of times the lock is attempted to be acquired before the process gives up control of the processor. *ustestlock* returns the instantaneous value of the lock, a 0 if it is not locked, and a 1 if it is locked. *usunsetlock* releases the lock.

When invoked with a valid lock, *ussetlock* and *uswsetlock* do not return until the lock is acquired. An invalid lock will yield unpredictable results.

It is allowed to call *usunsetlock* on a lock that is either not locked or locked by another process. In either case, the lock will be unlocked. Double tripping, i.e. calling a set lock function twice with the same lock is also permissible. The caller will block until another process unsets the lock.

These lock functions can only fail if the calling process has not yet registered with the shared arena. This occurs when related processes (via *sproc*(2)) start using previously allocated locks. A process can force itself to join an arena by issuing a *usinit*(3P) call. Once these new processes have successfully performed any lock, semaphore, or memory operation, these lock operations will never fail.

When using debugging lock types the following debugging prints can occur.

Double tripped on lock @ 0x... by pid ... will be printed when an attempt is made to acquire a lock that is already held by the caller.

Unlocking lock that other process lockd lock @ 0x... by pid ... will be printed when an attempt is made to release a lock that is not held by the process attempting to release it.

Unset lock, but lock not locked. lock @ 0x... pid ... will be printed when using debug software locks, and an attempt was made to unlock a lock that was not locked.

SEE ALSO

usconfig(3P), usinitlock(3P), usnewlock(3P), usfreelock(3P).

DIAGNOSTICS

ussetlock, uswsetlock, uscsetlock, and *ustestlock* will return a 1 if the lock is acquired and a 0 if the lock is not acquired. Otherwise, a -1 is returned and errno is set to indicate the error. *usunsetlock* returns 0 if the unlock succeeded, and -1 on error. With non-debug hardware locks, no errors can occur.

NAME

ustestsema – return the value of a semaphore

C SYNOPSIS

#include <ulocks.h>

int ustestsema (usema_t *sema);

FORTRAN SYNOPSIS

integer*4 function ustestsema (sema)
integer*4 sema

DESCRIPTION

ustestsema returns the current value of the semaphore specified by *sema*. This should be viewed as a snapshot only, useful for debugging.

SEE ALSO

usinitsema(3P), usnewsema(3P), uspsema(3P), uscpsema(3P), usvsema(3P).

DIAGNOSTICS

The current count of the semaphore is returned.

NAME

usvsema – frees a resource to a semaphore

C SYNOPSIS

#include <ulocks.h>

void usvsema (usema_t *sema);

FORTRAN SYNOPSIS

subroutine usvsema (sema)
integer*4 sema

DESCRIPTION

usvsema increments the counter associated with *sema*. If there are any processes queued waiting for the semaphore, the first one is awakened. *usvsema* uses *unblockproc*(2) to reactivate a suspended process.

SEE ALSO

unblockproc(2), usinitsema(3P), usnewsema(3P), uspsema(3P), uscpsema(3P).

DIAGNOSTICS

usvsema returns no value.

NAME

utimes – set file times

SYNOPSIS

#include <sys/time.h>

utimes(file, tvp)
char *file;
struct timeval tvp[2];

DESCRIPTION

The *utimes* call uses the ''accessed'' and ''updated'' times in that order from the *tvp* vector to set the corresponding recorded times for *file.*

The caller must be the owner of the file or the super-user. The ''inode-changed'' time of the file is set to the current time.

This routine emulates the 4.3BSD *utimes* system call.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error. *Utimes* will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[EINVAL] The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The process is not super-user and not the owner of the file.

[EACCES] Search permission is denied for a component of the path prefix.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *File* or *tvp* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading or writing the affected inode.

SEE ALSO

stat(2), utime(2)

NAME

vprintf, vfprintf, vsprintf – print formatted output of a variable argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vprintf (const char *format, va_list arg);

int vfprintf (FILE *stream, const char *format, va_list arg);

int vsprintf (char *s, const char *format, va_list arg);
```

DESCRIPTION

vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list, *arg*, as defined by *stdarg*(5). The *arg* parameter must be initialized by the *va_start* macro (and possibly subsequent *va_arg* calls). The *vprintf*, *vfprintf*, and *vsprintf* functions do not invoke the *va_end* macro.

EXAMPLE

The following demonstrates the use of *vfprintf* to write an error routine.

```
#include <stdarg.h>
#include <stdio.h>

/*
 *      error should be called as:
 *      error(function_name, format, arg1, arg2 ...);
 */
void
error(char *function_name, char *format, ...)
{
        va_list args;

        va_start(args, format);
        /* print out name of function causing error */
        fprintf(stderr, "ERROR in %s: ", function_name);
        /* print out remainder of message */
        vfprintf(stderr, format, args);
        va_end(args);
}
```

SEE ALSO

printf(3S), stdarg(5).

NAME

writev – write output gathered from buffers

SYNOPSIS

#include <sys/types.h>
#include <sys/uio.h>

cc = writev(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;

DESCRIPTION

Writev attempts to write to the object referenced by the descriptor *d*, gathering output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt−1]. The *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *Writev* will always write a complete area before proceeding to the next.

On objects capable of seeking, the *writev* starts at a position given by the pointer associated with *d*, see *lseek*(2). Upon return from *writev*, the pointer is incremented by the number of bytes actually written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

When using non-blocking I/O on objects such as sockets that are subject to flow control, *writev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Writev will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *D* is not a valid descriptor open for writing.

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process.

[EPIPE] An attempt is made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EFAULT] Part of *iov* to be written to the file points outside the process's allocated address space.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EAGAIN] The file was a *stream* marked for non-blocking I/O that could not accept data.

[EWOULDBLOCK]
The file was a socket marked for non-blocking I/O, and no data could be written immediately.

CAVEATS

Writev is implemented using *write*(2), and may ignore errors. If some data are written in the course of a *writev* call, but a *write* error occurs, the call returns the number of bytes successfully written, hiding the error. It is assumed that a subsequent call will discover persistent errors, and that sporadic errors such as EWOULDBLOCK can be ignored.

Writev attempts to copy data from *iov* into a buffer, in order to perform as few *writes* as possible. The buffer size is twice the value of **PIPE_MAX**, the maximum atomic write size for pipes defined in <*limits.h*>.

SEE ALSO

dup(2), fcntl(2), open(2), pipe(2), write(2), select(2), socket(2)

NAME

xdr – External Data Representation (XDR) library routines

SYNOPSIS AND DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)

```
XDR *xdrs;
char **arrp;
u_int *sizep, maxsize, elsize;
xdrproc_t elproc;
```

A filter primitive that translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_bool(xdrs, bp)

```
XDR *xdrs;
bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes(xdrs, sp, sizep, maxsize)

```
XDR *xdrs;
char **sp;
u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes()**, **xdr_opaque()** or **xdr_string()**.

void
xdr_destroy(xdrs)
XDR *xdrs;

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy()** is undefined.

xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;

A filter primitive that translates between C **double** precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;

A filter primitive that translates between C **enums** (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;

A filter primitive that translates between C **floats** and their external representations. This routine returns one if it succeeds, zero otherwise.

void
xdr_free(proc, objp)
xdrproc_t proc;
char *objp;

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

u_int
xdr_getpos(xdrs)
XDR *xdrs;

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;

A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to **long ***.

Warning: **xdr_inline()** may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;

A filter primitive that translates between C **long** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;

A filter primitive that translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR *xdrs;
char **objpp;
u_int objsize;
xdrproc_t xdrobj;

Like **xdr_reference()** execpt that it serializes NULL pointers, whereas **xdr_reference()** does not. Thus, **xdr_pointer()** can represent recursive data structures, such as binary trees or linked lists.

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize, recvsize;
char *handle;
int (*readit) (), (*writeit) ();

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the system calls **read** and **write**, except that *handle* is passed to the former routines as the first parameter. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;

This routine can be invoked only on streams created by **xdrrec_create()**. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof(xdrs)
XDR *xdrs;
int empty;

This routine can be invoked only on streams created by **xdrrec_create()**. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord(xdrs)
XDR *xdrs;

This routine can be invoked only on streams created by **xdrrec_create()**. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the *sizeof* the structure that **pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

Warning: this routine does not understand NULL pointers. Use **xdr_pointer()** instead.

xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from **xdr_getpos()**. This routine returns one if the XDR stream could be repositioned, and zero otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;

A filter primitive that translates between C **short** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

void
xdrstdio_create(xdrs, file, op)
    XDR *xdrs;
    FILE *file;
    enum xdr_op op;

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

Warning: the destroy routine associated with such XDR streams calls **fflush()** on the *file* stream, but never **fclose()**.

xdr_string(xdrs, sp, maxsize)
    XDR *xdrs;
    char **sp;
    u_int maxsize;

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_char(xdrs, ucp)
    XDR *xdrs;
    unsigned char *ucp;

A filter primitive that translates between **unsigned** C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_int(xdrs, up)
    XDR *xdrs;
    unsigned *up;

A filter primitive that translates between C **unsigned** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long(xdrs, ulp)
    XDR *xdrs;
    unsigned long *ulp;

A filter primitive that translates between C **unsigned long** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short(xdrs, usp)
XDR *xdrs;
unsigned short *usp;

A filter primitive that translates between C **unsigned short** integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
bool_t (*defaultarm) (); /* may equal NULL */

A filter primitive that translates between a discriminated C **union** and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an **enum_t**. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of **xdr_discrim()** structures. Each structure contains an ordered pair of [*value*,*proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the **xdr_discrim()** structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns one if it succeeds, zero otherwise.

xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR *xdrs;
char *arrp;
u_int size, elsize;
xdrproc_t elproc;

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_void()

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;

A primitive that calls
xdr_string(xdrs, sp,MAXUN.UNSIGNED);
where **MAXUN.UNSIGNED** is the maximum value of an unsigned integer. **xdr_wrapstring()** is handy because the RPC package passes a maximum of two XDR routines as parameters, and **xdr_string()**, one of the most frequently used primitives, requires three. Returns one if it succeeds, zero otherwise.

SEE ALSO

rpc(3R)

The *External Data Representation* chapter in the *Network Communications Guide*.